

第1章 概 论

本章要点

- 为什么要用编译器
- 与编译器相关的程序
- 翻译步骤
- 编译器中的主要数据结构
- 编译器结构中的其他问题
- 自举与移植
- TINY 样本语言与编译器
- C-Minus：编译器项目的一种语言

编译器是将一种语言翻译为另一种语言的计算机程序。编译器将源程序（source language）编写的程序作为输入，而产生用目标语言（target language）编写的等价程序。通常地，源程序为高级语言（high-level language），如C或C++，而目标语言则是目标机器的目标代码（object code，有时也称作机器代码（machine code）），也就是写在计算机机器指令中的用于运行的代码。这一过程可以用下图表示：

源程序 编译器 目标程序

编译器是一种相当复杂的程序，其代码的长度可从10 000行到1 000 000行不等。编写甚至读懂这样的一个程序都非易事，大多数的计算机科学家和专业人员也从来没有编写过一个完整的编译器。但是，几乎所有形式的计算均要用到编译器，而且任何一个与计算机打交道的专业人员都应掌握编译器的基本结构和操作。除此之外，计算机应用程序中经常遇到的一个任务就是命令解释程序和界面程序的开发，这比编译器要小，但使用的却是相同的技术。因此，掌握这一技术具有非常大的实际意义。

也正因为这一点，本书不仅仅要讲解基础知识，还为读者提供了所有必要的工具和设计编写真正的编译器的实践。要做到这些，就必须学习各项理论知识，而这主要应从自动机原理（它使编译器结构合理）着手。在讲述时我们假设读者并不了解自动机原理。当然，此处的观点与标准的自动机原理论著有所不同，这些论著特别强调编译过程；但是，学过自动机原理的读者就会发现对这些理论材料很熟悉，这部分阅读起来也十分迅速。特别是对于那些十分了解自动机原理背景的读者来说，对2.2节、2.3节、2.4节和3.2节就不必细读了。无论怎样，读者都应知道基本的数据结构和离散数学。机器结构和汇编语言的相关知识也很重要，在第8章“代码生成”中尤为如此。

实际编码技术的研究本身就要求认真规划，这是因为即使有很好的理论基础，编码的细节也可能会复杂得令人不知如何操作。本书包括了有关程序设计语言结构的一系列简单示例，并利用它们针对该项技术进行详细描述，讨论中使用到的语言被称作TINY。此外，附录A还提供了一个更广泛的示例，它包括了一个小小的但却非常复杂的适用于分类项目的C子集（称作C-Minus）。本书还有大量的练习，这其中包括简单的笔头训练、文本中的代码扩充，以及更多的相关编码练习。

总之，在编译器结构和被编译的程序设计语言的设计之间存在着一个很重要的交互。在本书中，只是附带着讲解了一下语言设计问题，而是着重于程序设计语言的概念和设计问题（参

见本章最后的“注意与参考”部分)。

首先将简要地介绍编译器的历史及其存在目的与理由, 以及与编译器相关的程序描述。接着讲解编译器的结构、各种翻译过程和相关的数据结构, 并联系一个简单的具体示例来示范这个结构。最后, 再概括地讲述一下编译器结构的其他问题, 这包括自举和移植, 以及本书后面用到的主要语言的描述。

1.1 为什么要用编译器

在本世纪40年代, 由于冯·诺伊曼在存储-程序计算机方面的先锋作用, 编写一串代码或程序已成必要, 这样计算机就可以执行所需的计算。开始时, 这些程序都是用机器语言(machine language)编写的。机器语言就是表示机器实际操作的数字代码, 例如:

```
C7 06 0000 0002
```

表示在IBM PC上使用的Intel 8x86处理器将数字2移至地址0000 (16进制) 的指令。当然, 编写这样的代码是十分费时和乏味的, 这种代码形式很快就被汇编语言(assembly language)代替了。在汇编语言中, 都是以符号形式给出指令和存储地址的。例如, 汇编语言指令

```
MOV X, 2
```

就与前面的机器指令等价(假设符号存储地址X是0000)。汇编程序(assembler)将汇编语言的符号代码和存储地址翻译成与机器语言相对应的数字代码。

汇编语言大大提高了编程的速度和准确度, 人们至今仍在使用着它, 在编码需要极快的速度和极高的简洁程度时尤为如此。但是, 汇编语言也有许多缺点: 编写起来也不容易, 阅读和理解很难; 而且汇编语言的编写严格依赖于特定的机器, 所以为一台计算机编写的代码在应用于另一台计算机时必须完全重写。很明显, 发展编程技术的下一个重要步骤就是以一個更类似于数学定义或自然语言的简洁形式来编写程序的操作, 它应与任何机器都无关, 而且也可由一个程序翻译为可执行的代码。例如, 前面的汇编语言代码可以写成一个简洁的与机器无关的形式

```
x = 2
```

起初人们担心这是不可能的, 或者即使可能, 目标代码也会因效率不高而没有多大用处。

在1954年至1957年期间, IBM的John Backus带领的一个研究小组对FORTRAN语言及其编译器的开发, 使得上面的担忧不必要了。但是, 由于当时处理中所涉及到的大多数程序设计语言的翻译并不为人所掌握, 所以这个项目的成功也伴随着巨大的辛劳。

几乎与此同时, 人们也在开发着第一个编译器, Noam Chomsky开始了他的自然语言结构的研究。他的发现最终使得编译器结构异常简单, 甚至还带有了一些自动化。Chomsky的研究导致了根据语言文法(grammar, 指定其结构的规则)的难易程度以及识别它们所需的算法来为语言分类。正如现在所称的——与乔姆斯基分类结构(Chomsky hierarchy)一样——包括了文法的4个层次: 0型、1型、2型和3型文法, 且其中的每一个都是其前者的专门化。2型(或上下文无关文法(context-free grammar))被证明是程序设计语言中最有用的, 而且今天它已代表着程序设计语言结构的标准方式。分析问题(parsing problem, 用于限定上下文无关语言的识别的有效算法)的研究是在60年代和70年代, 它相当完善地解决了这一问题, 现在它已是编译理论的一个标准部分。本书的第3、4和5章将研究上下文无关的语言和分析算法。

有穷自动机(finite automata)和正则表达式(regular expression)同上下文无关文法紧密相关, 它们与乔姆斯基的3型文法相对应。对它们的研究与乔姆斯基的研究几乎同时开始, 并且引出了表示程序设计语言的单词(或称为记号)的符号方式。第2章将讲述有穷自动机和正

则表达式。

人们接着又深化了生成有效的目标代码的方法，这就是最初的编译器，它们被一直使用至今。人们通常将其误称为优化技术（optimization technique），但因其从未真正地得到过被优化了的目标代码而仅仅改进了它的有效性，因此实际上应称作代码改进技术（code improvement technique）。第8章将讲述该技术的基础知识。

当分析问题变得好懂起来时，人们就在开发程序上花费了很大的功夫来研究这一部分的编译器的自动构造。这些程序最初被称为编译程序-编译器，但更确切地应称为分析程序生成器（parser generator），这是因为它们仅仅能够自动处理编译的一部分。这些程序中最著名的是 Yacc（yet another compiler-compiler），它是由 Steve Johnson 在 1975 年为 Unix 系统编写的，我们将在第 5 章中再次谈到它。类似地，有穷自动机的研究也发展了另一种称为扫描程序生成器（scanner generator）的工具，Lex（与 Yacc 同时，由 Mike Lesk 为 Unix 系统开发的）是这其中的佼佼者。读者将在第 2 章中学到 Lex。

在 70 年代后期和 80 年代早期，大量的项目都关注于编译器其他部分的生成自动化，这其中就包括了代码生成。这些尝试并未取得多少成功，这大概是因为操作太复杂而人们又对其不甚了解，本书也就不详细谈它了。

编译器设计最近的发展包括：首先，编译器包括了更为复杂的算法的应用程序，它用于推断和/或简化程序中的信息；这又与更为复杂的程序设计语言（可允许此类分析）的发展结合在一起。其中典型的有用于函数语言编译的 Hindley-Milner 类型检查的统一算法。其次，编译器已越来越成为基于窗口的交互开发环境（interactive development environment，IDE）的一部分，它包括了编辑器、链接程序、调试程序以及项目管理程序。这样的 IDE 的标准并没有多少，但是已沿着这一方向对标准的窗口环境进行开发了。这一专题的研究超出了本书的范围（但是读者可以参阅下一节中有关 IDE 部件的内容）。读者可以参阅本章末尾的“注意与参考”中的文献内容。尽管近年来对此进行了大量的研究，但是基本的编译器设计在近 20 年中都没有多大的改变，而且它们正迅速地成为计算机科学课程中的中心一环。

1.2 与编译器相关的程序

本节主要描述与编译器有关或专编译器一同使用的其他程序，以及那些在一个完整的语言开发环境中与编译器一同使用的程序（有一些已在前面提到过）。

(1) 解释程序（interpreter）

解释程序是如同编译器的一种语言翻译程序。它与编译器的不同之处在于：它立即执行源程序而不是生成在翻译完成之后才执行的目标代码。从原理上讲，任何程序设计语言都可被解释或被编译，但是根据所使用的语言和翻译情况，很可能会选用解释程序而不用编译器。例如，我们经常解释 BASIC 语言而不是去编译它。类似地，诸如 LISP 的函数语言也常常是被解释的。解释程序也经常用于教育和软件的开发，此处的程序很有可能被翻译若干次。而另一方面，当执行的速度是最为重要的因素时就使用编译器，这是因为被编译的目标代码比被解释的源代码要快得多，有时要快 10 倍或更多。但是，解释程序具有许多与编译器共享的操作，而两者之间也有一些混合之处。本书后面也将会提到解释程序，但重点仍是编译。

(2) 汇编程序（assembler）

汇编程序是用于特定计算机上的汇编语言的翻译程序。正如前面所提到的，汇编语言是计算机的机器语言的符号形式，它极易翻译。有时，编译器会生成汇编语言以作为其目标语言，然后再由一个汇编程序将它翻译成目标代码。

(3) 连接程序 (linker)

编译器和汇编程序都经常依赖于连接程序，它将分别在不同的目标文件中编译或汇编的代码收集到一个可直接执行的文件中。在这种情况下，目标代码，即还未被连接的机器代码，与可执行的机器代码之间就有了区别。连接程序还连接目标程序和用于标准库函数的代码，以及连接目标程序和由计算机的操作系统提供的资源（例如，存储分配程序及输入与输出设备）。有趣的是，连接程序现在正在完成编译器最早的一个主要活动（这也是“编译”一词的用法，即通过收集不同的来源来构造）。因为连接过程对操作系统和处理器有极大的依赖性，本书也就不研究它了。我们也对不细分连接的目标代码和可执行的代码，这是因为对于编译技术而言，这个区别并不重要。

(4) 装入程序 (loader)

编译器、汇编程序或连接程序生成的代码经常还不完全适用或不能执行，但是它们的主要存储器访问却可以在存储器的任何位置中且与一个不确定的起始位置相关。这样的代码被称为是可重定位的 (relocatable)，而装入程序可处理所有的与指定的基地址或起始地址有关的可重定位的地址。装入程序使得可执行代码更加灵活，但是装入处理通常是在后台（作为操作环境的一部分）或与连接相联合时才发生，装入程序极少会是实际的独立程序。

(5) 预处理器 (preprocessor)

预处理器是在真正的翻译开始之前由编译器调用的独立程序。预处理器可以删除注释、包含其他文件以及执行宏（宏 macro 是一段重复文字的简短描写）替代。预处理器可由语言（如 C）要求或以后作为提供额外功能（诸如为 FORTRAN 提供 Ratfor 预处理器）的附加软件。

(6) 编辑器 (editor)

编译器通常接受由任何生成标准文件（例如 ASCII 文件）的编辑器编写的源程序。最近，编译器已与另一个编辑器和其他程序捆绑进一个交互的开发环境——IDE 中。此时，尽管编辑器仍然生成标准文件，但会转向正被讨论的程序设计语言的格式或结构。这样的编辑器称为基于结构的 (structure based)，且它早已包括了编译器的某些操作；因此，程序员就会在程序的编写时而不是在编译时就得知错误了。从编辑器中也可调用编译器以及与之共用的程序，这样程序员无需离开编辑器就可执行程序。

(7) 调试程序 (debugger)

调试程序是可在被编译了的程序中判定执行错误的程序，它也经常与编译器一起放在 IDE 中。运行一个带有调试程序的程序与直接执行不同，这是因为调试程序保存着所有的或大多数源代码信息（诸如行数、变量名和过程）。它还可以在预先指定的位置（称为断点 (breakpoint)）暂停执行，并提供有关已调用的函数以及变量的当前值的信息。为了执行这些函数，编译器必须为调试程序提供恰当的符号信息，而这有时却相当困难，尤其是在一个要优化目标代码的编译器中。因此，调试又变成了一个编译问题，本书的内容就不涉及它了。

(8) 描述器 (profiler)

描述器是在执行中搜集目标程序行为统计的程序。程序员特别感兴趣的统计是每一个过程的调用次数和每一个过程执行时间所占的百分比。这样的统计对于帮助程序员提高程序的执行速度极为有用。有时编译器也甚至无需程序员的干涉就可利用描述器的输出来自动改进目标代码。

(9) 项目管理程序 (project manager)

现在的软件项目通常大到需要由一组程序员来完成，这时对那些由不同人员操作的文件进行整理就非常重要了，而这正是项目管理程序的任务。例如，项目管理程序应将由不同的程序

员制作的文件的各个独立版本整理在一起，它还应保存一组文件的更改历史，这样就能维持一个正在开发的程序的连贯版本了（这对那些由单个程序员管理的项目也很有用）。项目管理程序的编写可与语言无关，但当其与编译器捆绑在一起时，它就可以保持有关特定的编译器和建立一个完整的可执行程序的链接程序操作的信息。在 Unix 系统中有两个流行的项目管理程序：scs（source code control system）和 rcs（revision control system）。

1.3 翻译步骤

编译器内部包括了许多步骤或称为阶段（phase），它们执行不同的逻辑操作。将这些阶段设想为编译器中一个个单独的片断是很有用的，尽管在应用中它们是经常组合在一起的，但它们确实是作为单独的代码操作来编写的。图 1-1 是编译器中的阶段和与以下阶段（文字表、符号表和错误处理器）或其中的一部分交互的3个辅助部件。这里只是简要地描述一下每个阶段，今后大家还会更详细地学到它们（文字表和符号表在 1.4 节中，错误处理器在 1.5 节）。

(1) 扫描程序（scanner）

在这个阶段编译器实际阅读源程序（通常以字符流的形式表示）。扫描程序执行词法分析（Lexical analysis）：它将字符序列收集到称作记号（token）的有意义单元中，记号同自然语言，如英语中的字词相似。因此可以认为扫描程序执行与拼写相似的任务。

例如在下面的代码行（它可以是 C 程序的一部分）中：

```
a[index] = 4 + 2
```

这个代码包括了 12 个非空字符，但只有 8 个记号：

a	标识符
[左括号
index	标识符
]	右括号
=	赋值
4	数字
+	加号
2	数字

每一个记号均由一个或多个字符组成，在进一步处理之前它已被收集在一个单元中。

扫描程序还可完成与识别记号一起执行的其他操作。例如，它可将标识符输入到符号表中，将文字（literal）输入到文字表中（文字包括诸如 3.1415926535 的数字常量，以及诸如“Hello, world!”的引用字符串）。

(2) 语法分析程序（parser）

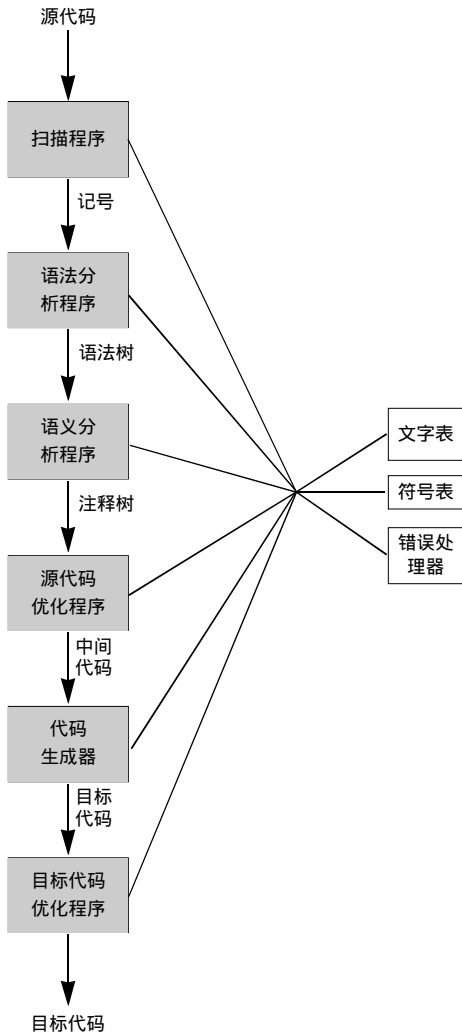
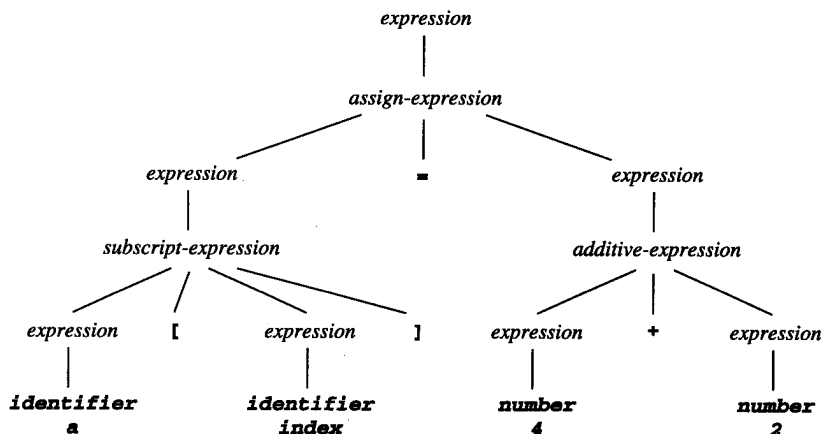


图1-1 编译器的阶段

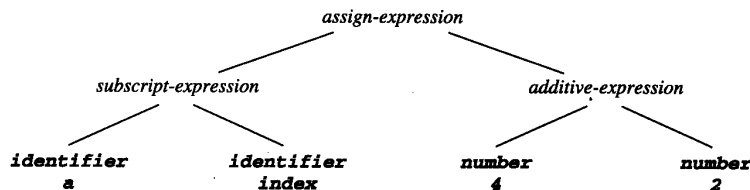
语法分析程序从扫描程序中获取记号形式的源代码，并完成定义程序结构的语法分析 (syntax analysis)，这与自然语言中句子的语法分析类似。语法分析定义了程序的结构元素及其关系。通常将语法分析的结果表示为分析树 (parse tree) 或语法树 (syntax tree)。

例如，还是那行 C 代码，它表示一个称为表达式的结构元素，该表达式是一个由左边为下标表达式、右边为整型表达式的赋值表达式组成。这个结构可按下面的形式表示为一个分析树：



请注意，分析树的内部节点均由其表示的结构名标示出，而分析树的叶子则表示输入中的记号序列（结构名以不同字体表示以示与记号的区别）。

分析树对于显示程序的语法或程序元素很有帮助，但是对于表示该结构却显得力不从心了。分析程序更趋向于生成语法树，语法树是分析树中所含信息的浓缩（有时因为语法树表示从分析树中的进一步抽取，所以也被称为抽象的语法树 (abstract syntax tree)）。下面是一个 C 赋值语句的抽象语法树的例子：



请注意，在语法树中，许多节点（包括记号节点在内）已经消失。例如，如果知道表达式是一个下标运算，则不再需要用括号 “[” 和 “] ” 来表示该操作是在原始输入中。

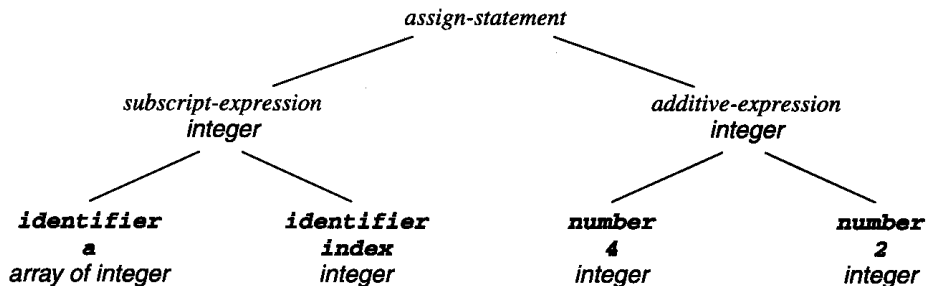
(3) 语义分析程序 (semantic analyzer)

程序的语义就是它的“意思”，它与语法或结构不同。程序的语义确定程序的运行，但是大多数的程序设计语言都具有在执行之前被确定而不易由语法表示和由分析程序分析的特征。这些特征被称作静态语义 (static semantic)，而语义分析程序的任务就是分析这样的语义（程序的“动态”语义具有只有在程序执行时才能确定的特性，由于编译器不能执行程序，所以它不能由编译器来确定）。一般的程序设计语言的典型静态语义包括声明和类型检查。由语义分析程序计算的额外信息（诸如数据类型）被称为属性 (attribute)，它们通常是作为注释或“装饰”增加到树中（还可将属性添加到符号表中）。

在正运行的 C 表达式

```
a [index] = 4 + 2
```

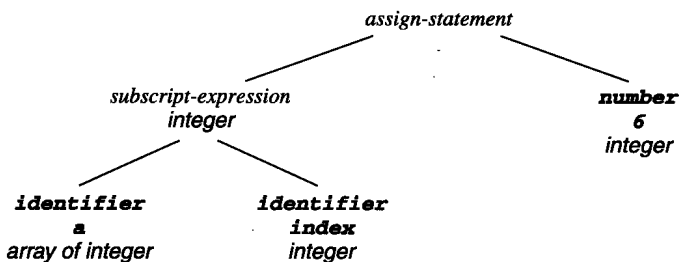
中，该行分析之前收集的典型类型信息可能是：**a**是一个整型值的数组，它带有来自整型子范围的下标；**index**则是一个整型变量。接着，语义分析程序将用所有的子表达式类型来标注语法树，并检查赋值是否使这些类型有意义了，如若没有，则声明一个类型匹配错误。在上例中，所有的类型均有意义，有关语法树的语义分析结果可用以下注释了的树来表示：



(4) 源代码优化程序 (source code optimizer)

编译器通常包括许多代码改进或优化步骤。绝大多数最早的优化步骤是在语义分析之后完成的，而此时代码改进可能只依赖于源代码。这种可能性是通过将这一操作提供为编译过程中的单独阶段指出的。每个编译器不论在已完成的优化种类方面还是在优化阶段的定位中都有很大的差异。

在上例中，我们包括了一个源代码层次的优化机会，也就是：表达式 $4+2$ 可由编译器计算先得到结果 6 （这种优化称为常量合并 (constant folding)）。当然，还会有更复杂的情况（有些将在第8章中提到）。还是在上例中，通过将根节点右面的子树合并为它的常量值，这个优化就可以直接在（注释）语法树上完成：



尽管许多优化可以直接在树上完成，但是在很多情况下，优化接近于汇编代码线性化形式的树更为简便。这样节点的变形有许多，但是三元式代码 (three-address code)（之所以这样称呼是因为它在存储器中包含了3个（或3个以上）位置的地址）却是标准选择。另一个常见的选择是P-代码 (P-code)，它常用于Pascal编译器中。

在前面的例子中，原先的C表达式的三元式代码应是：

```
t = 4 + 2
a [ index ] = t
```

（请注意，这里利用了一个额外的临时变量 **t** 存放加法的中间值）。这样，优化程序就将这个代码改进为两步。首先计算加法的结果：

```
t = 6
a [ index ] = t
```

接着，将 t 替换为该值以得到三元语句

```
a[index] = 6
```

图1-1已经指出源代码优化程序可能通过将其输出称为中间代码（intermediate code）来使用三元式代码。中间代码一直是指一种位于源代码和目标代码（例如三元式代码或类似的线性表示）之间的代码表示形式。但是，我们可以更概括地认为它是编译器使用的源代码的任何一个内部表示。此时，也可将语法树称作中间代码，源代码优化程序则确实能继续在其输出中使用这个表示。有时，这个中间代码也称作中间表示（intermediate representation, IR）。

(5) 代码生成器（code generator）

代码生成器得到中间代码（IR），并生成目标机器的代码。尽管大多数编译器直接生成目标代码，但是为了便于理解，本书用汇编语言来编写目标代码。正是在编译的这个阶段中，目标机器的特性成为了主要因素。当它存在于目标机器时，使用指令不仅是必须的而且数据的形式表示也起着重要的作用。例如，整型数据类型的变量和浮点数据类型的变量在存储器中所占的字节数或字数也很重要。

在上面的示例中，现在必须决定怎样存储整型数来为数组索引生成代码。例如，下面是所给表达式的一个可能的样本代码序列（在假设的汇编语言中）：

```
MOV      R0, index      ;; value of index -> R0
MUL      R0, 2           ;; double value in R0
MOV      R1, &a         ;; address of a -> R1
ADD      R1, R0          ;; add R0 to R1
MOV      *R1, 6          ;; constant 6 -> address in R1
```

在以上代码中，为编址模式使用了一个类似C的协定，因此 $\&a$ 是 a 的地址（也就是数组的基地址）， $*R1$ 则意味着间接寄存器地址（因此最后一条指令将值6存放在 $R1$ 包含的地址中）。这个代码还假设机器执行字节编址，并且整型数占据存储器的两个字节（所以在第2条指令中用2作为乘数）。

(6) 目标代码优化程序（target code optimizer）

在这个阶段中，编译器尝试着改进由代码生成器生成的目标代码。这种改进包括选择编址模式以提高性能、将速度慢的指令更换成速度快的，以及删除多余的操作。

在上面给出的样本目标代码中，还可以做许多更改：在第2条指令中，利用移位指令替代乘法（通常地，乘法很费时间），还可以使用更有效的编址模式（例如用索引地址来执行数组存储）。使用了这两种优化后，目标代码就变成：

```
MOV R0, index      ;; value of index -> R0
SHL R0             ;; double value in R0
MOV &a[R0], 6       ;; constant 6 -> address a + R0
```

到这里，对编译器阶段的简要描述就结束了，但我们还应特别强调这些讲述仅仅是示意性的，也无需表示出正在工作中的编译器的实际结构。编译器在其结构细节上确实差别很大，然而，上面讲到的阶段总会出现在几乎所有的编译器的某个形式上。

我们还谈到了用于维持每一个阶段所需信息的数据结构，例如语法树、中间代码（假设它们并不相同）、文字表和符号表。下一节是编译器中的主要数据结构的概览。

1.4 编译器中的主要数据结构

当然，由编译器的阶段使用的算法与支持这些阶段的数据结构之间的交互是非常强大的。编译器的编写者尽可能有效实施这些方法且不引起复杂性。理想的情况是：与程序大小成线性

比例的时间内编译器，换言之就是，在 $O(n)$ 时间内， n 是程序大小的度量（通常是字符数）。本节将讲述一些主要的数据结构，它们是其操作部分阶段所需要的，并用来在阶段中交流信息。

(1) 记号 (token)

当扫描程序将字符收集到一个记号中时，它通常是以符号表示这个记号；这也就是说，作为一个枚举数据类型的值来表示源程序的记号集。有时还必须保留字符串本身或由此派生出的其他信息（例如：与标识符记号相关的名字或数字记号值）。在大多数语言中，扫描程序一次只需要生成一个记号（这称为单符号先行（single symbol lookahead））。在这种情况下，可以用全程变量放置记号信息；而在别的情况（最为明显的是 FORTRAN）下，则可能会需要一个记号数组。

(2) 语法树 (syntax tree)

如果分析程序确实生成了语法树，它的构造通常为基于指针的标准结构，在进行分析时动态分配该结构，则整棵树可作为一个指向根节点的单个变量保存。结构中的每一个节点都是一个记录，它的域表示由分析程序和之后的语义分析程序收集的信息。例如，一个表达式的数据类型可作为表达式的语法树节点中的域。有时为了节省空间，这些域也是动态分配或存放在诸如符号表的其他数据结构中，这样就可以有选择地进行分配和释放。实际上，根据它所表示的语言结构的类型（例如：表达式节点对于语句节点或声明节点都有不同的要求），每一个语法树节点本身都可能要求存储不同的属性。在这种情况下，可由不同的记录表示语法树中的每个节点，每个节点类型只包含与本身相关的信息。

(3) 符号表 (symbol table)

这个数据结构中的信息与标识符有关：函数、变量、常量以及数据类型。符号表几乎与编译器的所有阶段交互：扫描程序、分析程序或将标识符输入到表格中的语义分析程序；语义分析程序将增加数据类型和其他信息；优化阶段和代码生成阶段也将利用由符号表提供的信息选出恰当的代码。因为对符号表的访问如此频繁，所以插入、删除和访问操作都必须比常规操作更有效。尽管可以使用各种树的结构，但杂凑表却是达到这一要求的标准数据结构。有时在一个列表或栈中可使用若干个表格。

(4) 常数表 (literal table)

常数表的功能是存放在程序中用到的常量和字符串，因此快速插入和查找在常数表中也十分重要。但是，在其中却无需删除，这是因为它的数据全程应用于程序而且常量或字符串在该表中只出现一次。通过允许重复使用常量和字符串，常数表对于缩小程序在存储器中的大小显得非常重要。在代码生成器中也需要常数表来构造用于常数和目标代码文件中输入数据定义的符号地址。

(5) 中间代码 (intermediate code)

根据中间代码的类型（例如三元式代码和 P-代码）和优化的类型，该代码可以是文本串的数组、临时文本文件或结构的连接列表。对于进行复杂优化的编译器，应特别注意选择允许简单重组的表示。

(6) 临时文件 (temporary file)

计算机过去一直未能在编译器时将整个程序保留在存储器中。这一问题已经通过使用临时文件来保存翻译时中间步骤的结果或通过“匆忙地”编译（也就是只保留源程序早期部分的足够信息用以处理翻译）解决了。存储器的限制现在也只是一个小问题了，现在可以将整个编译单元放在存储器之中，特别是在可以分别编译的语言中时。但是偶尔还是会发现需要在某些运

行步骤中生成中间文件。其中典型的是代码生成时需要反填（backpatch）地址。例如，当翻译如下的条件语句时

```
if x = 0 then ... else ...
```

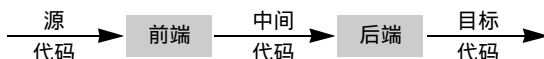
在知道else部分代码的位置之前必须由文本跳到else部分：

```
CMP X, 0
JNE NEXT ;; location of NEXT not yet known
< code for then-part >
NEXT:
< code for else-part >
```

通常，必须为NEXT的值留出一个空格，一旦知道该值后就会将该空格填上，利用临时文件可以很容易地做到这一点。

1.5 编译器结构中的其他问题

可从许多不同的角度来观察编译器的结构。1.3节已讲述了它的阶段——用来表示编译器的逻辑结构。此外，还有其他一些可能的观点：编译器的物理结构、操作的顺序等等。由于编译器的结构对其可靠性、有效性、可用性以及可维护性都有很大的影响，所以编译器的编写者应熟悉尽可能多的有关编译器结构的观点。本节将考虑编译器结构的其他方面以及每一个观点是如何应用的。



(1) 分析和综合

在这个观点中，已将分析源程序以计算其特性的编译器操作归为编译器的分析（analysis）部分，而将生成翻译代码时所涉及到的操作称作编译器的综合（synthesis）部分。当然，词法分析、语法分析和语义分析均属于分析部分，而代码生成却是综合部分。在优化步骤中，分析和综合都有。分析正趋向于易懂和更具有数学性，而综合则要求更深的专业技术。因此，将分析步骤和综合步骤两者区分开来以便发生变化时互不影响是很有用的。

(2) 前端和后端

本观点认为，将编译器分成了只依赖于源语言（前端（front end））的操作和只依赖于目标语言（后端（back end））的操作两部分。这与将其分成分析和综合两部分是类似的：扫描程序、分析程序和语义分析程序是前端，代码生成器是后端。但是一些优化分析可以依赖于目标语言，这样就是属于后端了，然而中间代码的综合却经常与目标语言无关，因此也就属于前端了。在理想情况下，编译器被严格地分成这两部分，而中间表示则作为其间的交流媒介。

这一结构对于编译器的可移植性（portability）十分重要，此时设计的编译器既能改变源代码（它涉及到重写前端），又能改变目标代码（它还涉及到重写后端）。在实际中，这是很难做到的，而且称作可移植的编译器仍旧依赖于源语言和目标语言。其部分原因是程序设计语言和机器构造的快速发展以及根本性的变化，但是有效地保持移植一个新的目标语言所需的信息或使数据结构普遍地适合改变为一个新的源语言所需的信息却十分困难。然而人们不断分离前端和后端的努力会带来更方便的可移植性。

(3) 遍

编译器发现，在生成代码之前多次处理整个源程序很方便。这些重复就是遍（pass）。首遍是从源中构造一个语法树或中间代码，在它之后的遍是由处理中间表示、向它增加信息、更

换结构或生成不同的表示组成。遍可以和阶段相应，也可无关——遍中通常含有若干个阶段。实际上，根据语言的不同，编译器可以是一遍（one pass）——所有的阶段由一遍完成，其结果是编译得很好，但（通常）代码却不太有效。Pascal语言和C语言均允许单遍编译。（Modula-2语言的结构则要求编译器至少有两遍）。大多数带有优化的编译器都需要超过一遍：典型的安排是将一遍用于扫描和分析，将另一遍用于语义分析和源代码层优化，第3遍用于代码生成和目标层的优化。更深层的优化则可能需要更多的遍：5遍、6遍、甚至8遍都是可能的。

(4) 语言定义和编译器

我们注意到在1.1节中，程序设计语言的词法和语法结构通常用形式的术语指定，并使用正则表达式和上下文无关文法。但是，程序设计语言的语义通常仍然是由英语（或其他的自然语言）描述的。这些描述（与形式的词法及语法结构一起）一般是集中在一个语言参考手册（language reference manual）或语言定义（language definition）之中。因为编译器的编写者掌握的技术对于语言的定义有很大的影响，所以在使用了一种新的语言之后，语言的定义和编译器同时也能够得到开发。类似地，一种语言的定义对于构造编译器所需的技术也有很大的关系。

编译器的编写者更经常遇到的情况是：正在实现的语言是众所周知的并已有了语言定义。有时这个语言定义已达到了某个语言标准（language standard）的层次，语言标准是指得到诸如美国国家标准协会（American National Standards Institute, ANSI）或国际标准化组织（International Organization for Standardization, ISO）的官方标准组织批准的标准。FORTRAN、Pascal和C语言就具有ANSI标准，Ada有一个通过了美国政府批准的标准。在这种情况下，编译器的编写者必须解释语言的定义并执行符合语言定义的编译器。通常做到这一点并不容易，但是有时由于有了标准测试程序集（测试组（test suite）），就能够测试编译器（Ada有一个测试组），这又变得简单起来了。文本中使用的TINY示范语言有其词法、语法和语义结构，在2.5节、3.7节和6.5节中将分别谈到这些。附录A包括了用于C-Minus编译器项目语言的一个最小的语言参考手册。

有时候，一种语言可从数学术语的形式定义（formal definition）中得到它的语义。现在人们已经使用了许多方法，尽管一个称作表示语义（denotational semantics）的方法已经成为较为常用的方法，在函数编程共同体中尤为如此，但现在仍然没有一种可成为标准的方法。当语言有一个形式定义时，那么在理论上就有可能给出编译器与该定义一致的数学证明，但是由于这太难了而几乎从未有人做过。无论怎样，该技术已超出了本书的范围，本书也不会涉及到形式语义方面的知识。

运行时环境的结构和行为是尤其受到语言定义影响的编译器构造的一个方面。运行时环境将在第7章中学习。尽管此时它没有多大用处，但程序设计语言所允许的数据结构（尤其是被许可的函数调用和返回值的类型）对于运行时系统的复杂程度具有决定性意义。以下是运行时环境的3个基本类型（按难易程度排列）：

首先是FORTRAN77，它没有指针或动态分配，也没有递归函数调用，但它允许有一个完整的静态运行时环境。在这个环境中，所有存储器的分配都在执行之前进行。因为无需生成代码来维护环境，编译器的编写者的分配工作也就容易许多了。其次是Pascal、C和其他类似Algol的语言，它们允许有限动态分配以及递归函数的调用，并且要求“半动态”或带有额外的动态结构（称为堆，由此程序员可安排动态分配）的基于栈的运行时环境。最后是面向对象的函数语言，如LISP和Smalltalk，它们要求“完全动态”的环境，在其中所有的分配都是由编译器的生成代码自动完成的。因为它要求也能够自动释放存储器，而这又相应地要求复杂的

“垃圾回收”算法，所以它很复杂。在学习运行时环境时将会讨论到它，但是更为复杂的内容就超出本书的范围了。

(5) 编译器的选项和界面

编译器结构的一个重要方面是包含了一种机制与操作系统相连接，并为了满足用户的各种目的而提供选择。界面机制的例子是提供对目标机器的文件系统的访问以及输入和输出功能。用户的选项可包括列表特征（长度、出错信息和相互对照表）的说明和代码优化选项（只是某个优化的执行）。选项和界面共称为编译器的语用学（pragmatics）。有时一种语言的定义指出必须提供的语用学。例如，Pascal和C语言均指出一定的输入/输出过程（在Pascal中，它们是语言特性中的一部分；而在C语言中，它们是标准库说明的一个部分）。在Ada中，许多编译器的指示（称为（pragmas））则是语言定义的一部分。例如：Ada语句

```
pragma LIST(ON);  
...  
pragma LIST(OFF);
```

为包含在pragmas中的程序部分生成了一个编译器列表。在这个文本中，我们发现编译器指示仅存在于用作编译器调试的生成列表信息的上下文中；另外，也不会在输入/输出和操作系统界面中处理问题，这是因为它们涉及到了大量的细节，而且随操作系统的不同而有很大差异。

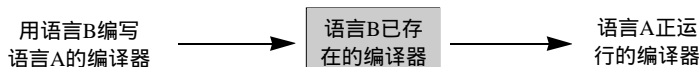
(6) 出错处理

编译器的一个最为重要的功能是其对源程序中错误的反应。几乎在编译的每一个阶段中都可以诊断出错误来。这些静态（或称为编译时（compile-time））的错误（static error）必须由编译器来报告，而编译器能够生成有意义的出错信息并在每一个错误之后恢复编译是非常重要的。编译器的每一个阶段都需要一个类型略为不同的出错处理，因此错误处理器（error handler）必须包括不同的操作，每个操作都给出指定的阶段和结构。因此，读者将在相应的章节中学到每一个阶段的出错处理技术。

语言定义经常要求编译器不仅能够找到静态错误，而且还能找到执行错误。这就需要编译器生成额外的代码，该代码将执行恰当的运行时测试，以保证所有这样的错误都将在执行时引起一个合适的事件。在此类事件中，最简单的就是中止程序的执行。但这经常是不合适的，而且语言的定义可能要求存在异常处理（exception handling）机制。这将使运行时系统的管理变得非常复杂，当程序可能由错误发生处继续执行时尤其如此。本书并不涉及到这样一个机制的执行情况，但会提到编译器如何生成测试代码，以保证指定的运行时错误引起执行中止。

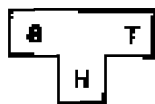
1.6 自举与移植

前面已经讨论过源语言和目标语言在编译器结构中的决定因素，以及将源语言和目标语言分为前端和后端的作用，但是却未提到过编译器构造过程中涉及到的另一个语言：编写编译器本身使用的语言。为了使编译器能立即执行，这个执行（或宿主（host））语言只能是机器语言。当时并没有编译器，因此这确实是最初的编译器编写所用的语言。现在更为合理的方法是用另一种语言来编写编译器，而使用该种语言的编译器早已存在了。如果现存的编译器已经在目标机器上运行了，则只需利用现存的编译器编译出新的编译器以得到可运行的程序：



当语言B的现存编译器没有运行在目标机器上时，情况会更复杂一些。编译将产生一个交叉编

译器 (cross compiler), 也就是一个为不同于它在运行之上的机器生成目标代码的编译器。这种以及其他更为复杂的情况最好通过将编译器画成一个 T 型图 (T-diagram) (以其形状来命名) 来描述。用语言 H (代表宿主语言) 编写的编译器将语言 S (代表源语言) 翻译为语言 T (代表目标语言) 可画成以下的 T 型图:

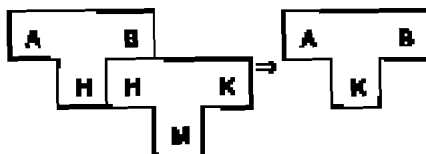


请注意, 这与表示编译器是在“机器”H上运行是等价的 (如果 H 不是机器代码, 则可认为其是一个假定机器的可执行代码)。我们一般都希望 H 与 T 相同 (也就是编译器为与之运行之上同样的机器生成代码), 但是也并不是必须这样做。

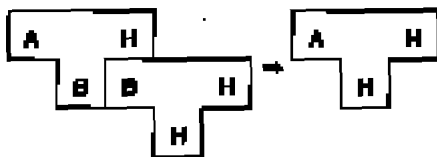
可以用两种方法组合 T 型图。一种是, 如果在一台机器 H 上运行有两个编译器, 其中一个编译器将语言 A 翻译为语言 B, 另一个将语言 B 翻译为语言 C, 就可按照将第 1 个的输出作为第 2 个的输入来组合。其所得结果就是一个在机器 H 上的由 A 到 C 的编译。将该过程表示为:



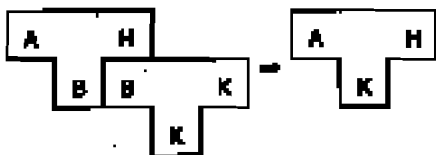
另一种是, 利用由“机器”H到“机器”K的编译器来翻译由 H 到 K 的其他编译器的执行语言。表示如下:



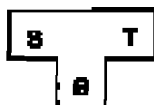
在上面的描述中, 第 1 个假定是, 在机器 H 上利用语言 B 现存的编译器将语言 A 翻译为用 B 编写的语言 H。它是前面所讲的特例, 如下所示:



第 2 个假定是, 当语言 B 的编译器运行在另一台机器上时, 就会引出语言 A 的交叉编译器。如下图所示:



以将要编译的相同语言编写一个编译器是很普通的:



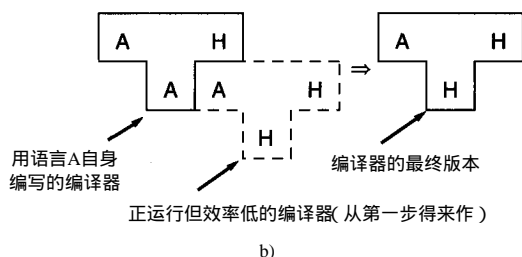
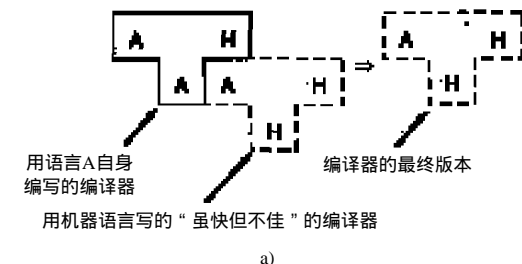


图1-2 自举进程

a) 自举进程中的第1个步骤 b) 自举进程中的第2个步骤

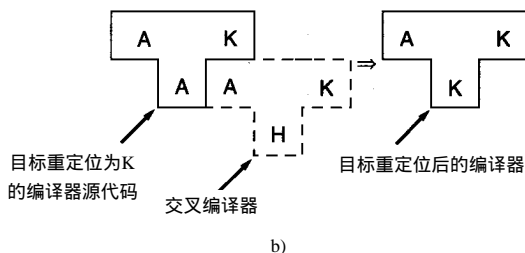
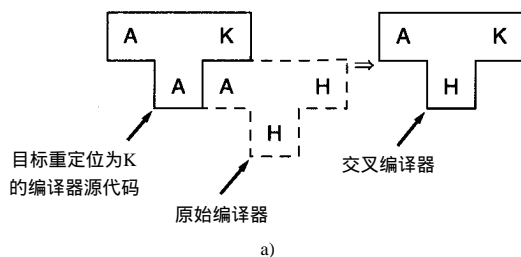


图1-3 移植一个在其自身源代码中编写的编译器

a) 步骤1 b) 步骤2

但这将表现为一个循环错误：因为如果源语言的编译器不存在，那么编译器本身也就不可能被编译了。从这个方法中可以得到很重要的启示。

让我们设想一个问题：如何解决循环。我们可以在汇编语言中编写一个“虽快但不佳”的编译器，并翻译那些在编译器中真正使用得到的语言特征（当然，在编写“较好的”编译器时，会对使用那些特征有所限制）。这些“虽快但不佳”的编译器也可能产生极为无效的代码（它仅需要正确而已！）。一旦运行这个“虽快但不佳”的编译器，就可用它来编译那个“较好的”编译器。接着，对“较好的”编译器进行重编译以得到最终的版本。人们将这个过程称为自举（bootstrapping）。图1-2a和图1-2b描述了这一过程。

自举之后，在源代码和执行代码中就有了一个编译器。这样做的好处在于：通过应用与前面相同的两步过程，编译器的源代码的任何改进都会立即被自举到一个正在工作着的编译器中。

除此之外，还有一个好处。现在将编译器移植到一个新的主机，只要求重写源代码的后端来生成新机器的代码。接着用旧的编译器来编译它以生成一个交叉编译器，该编译器又再次被交叉编译器重新编译，以得到新机器的版本。图1-3a和图1-3b描述了这一过程。

1.7 TINY样本语言与编译器

任何一本关于编译结构的书如果不包括编译过程步骤的示例就不能算完整。本书将会多次用从现有的语言（如C、C++、Pascal和Ada）中抽取的实例来讲解。但是仅用这些实例来描述编译器的各个部分是如何协调一致的却不够。因此，写出一个完整的编译器并对其操作进行注释仍是很必要的。

描述真实的编译器非常困难。“真正的”编译器——也就是希望在每天编程中用到的——内容太复杂而且不易在本教材中掌握。另一方面，一种很小的语言（其列表包括10页左右的文本）的编译也不可能准确地描述出“真正的”编译器所需的所有特征。

为了解决上述问题，人们在（ANSI）C中为小型语言提供了完整的源代码，一旦能明白这

种技术，就能够很容易地理解这种小型语言的编译器了。这种语言称作 TINY，在每一章的示例中都会用到它，它的编译代码也很快会被提到。完整的编译代码汇集在附录 B 中。

还有一个问题是：选择哪一种机器语言作为 TINY 编译器的目标语言？为现有的处理器使用真正的机器代码的复杂性使得这个选择很困难。但是选择特定的处理器也将影响到执行这些机器生成的目标代码。相反地，可将目标代码简化为一个假定的简单处理器的汇编语言，这个处理器称为 TM 机（tiny machine）。在这里只是简单地谈谈，详细内容将放在第 8 章（代码生成）中。附录 C 有 C 的 TM 模拟程序列表。

1.7.1 TINY 语言

TINY 的程序结构很简单，它在语法上与 Ada 或 Pascal 的语法相似：仅是一个由分号分隔开的语句序列。另外，它既无过程也无声明。所有的变量都是整型变量，通过对其赋值可较轻易地声明变量（类似 FORTRAN 或 BASIC）。它只有两个控制语句：if 语句和 repeat 语句，这两个控制语句本身也可包含语句序列。If 语句有一个可选的 else 部分且必须由关键字 end 结束。除此之外，read 语句和 write 语句完成输入/输出。在花括号中可以有注释，但注释不能嵌套。

TINY 的表达式也局限于布尔表达式和整型算术表达式。布尔表达式由对两个算术表达式的比较组成，该比较使用 < 与 = 比较算符。算术表达式可以包括整型常数、变量、参数以及 4 个整型算符 +、-、*、/，此外还有一般的数学属性。布尔表达式可能只作为测试出现在控制语句中——而没有布尔型变量、赋值或 I/O。

程序清单 1-1 是该语言中的一个阶乘函数的简单编程示例。这个例子在整本书中都会用到。

程序清单 1-1 一个输出其输入阶乘的 TINY 语言程序

```
{ Sample program
  in TINY language -
  computes factorial
}
read x; { input an integer }
if x > 0 then { don't compute if x <= 0 }
  fact := 1;
  repeat
    fact := fact * x;
    x := x - 1
  until x = 0;
  write fact { output factorial of x }
end
```

虽然 TINY 缺少真正程序设计语言所需要的许多特征——过程、数组且浮点值是一些较大的省略——但它足可以用来例证编译器的主要特征了。

1.7.2 TINY 编译器

TINY 编译器包括以下的 C 文件，（为了包含而）把它的头文件放在左边，它的代码文件放在右边：

globals.h	main.c
util.h	util.c
scan.h	scan.c
parse.h	parse.c

<code>syntab.h</code>	<code>syntab.c</code>
<code>analyze.h</code>	<code>analyze.c</code>
<code>code.h</code>	<code>code.c</code>
<code>cgen.h</code>	<code>cgen.c</code>

除了将`main.c`放在`globals.h`的前面之外，这些文件的源代码及其行号都按顺序列在附录B中了。任何代码文件都包含了`globals.h`头文件，它包括了数据类型的定义和整个编译器均使用的全程变量。`main.c`文件包括运行编译器的主程序，它还分配和初始化全程变量。其他的文件则包含了头/代码文件对，在头文件中给出了外部可用的函数原型以及在相关代码文件中的实现（包括静态局部函数）。`scan`、`parse`、`analyze`和`cgen`文件与图1-1中的扫描程序、分析程序、语义分析程序和代码生成器各阶段完全相符。`util`文件包括了实用程序函数，生成源代码（语法树）的内部表示和显示列表与出错信息均需要这些函数。`syntab`文件包括执行与TINY应用相符的符号表的杂凑表。`code`文件包括用于依赖目标机器（将在1.7.3节描述的TM机）的代码生成的实用程序。图1-1还缺少一些其他部分：没有单独的错误处理器或文字表且没有优化阶段；没有从语法树上分隔出来的中间代码；另外，符号表只与语义分析程序和代码生成器交互（这将在第6章中再次讨论到）。

虽然这些文件中的交互少了，但是编译器仍有4遍：第1遍由构造语法树的扫描程序和分析程序组成；第2遍和第3遍执行语义分析，其中第2遍构造符号表而第3遍完成类型检查；最后一遍是代码生成器。在`main.c`中驱动这些遍的代码十分简单。当忽略了标记和编辑时，它的中心代码如下（请参看附录B中的第69、77、79和94行）：

```
syntaxTree = parse( );
buildSyntab (syntaxTree);
typeCheck (syntaxTree);
codeGen (syntaxTree, codefile);
```

为了灵活起见，我们还编写了条件编译标志，以使得有可能创建出一部分的编译器。如下是该标志及其效果：

标 志	设置效果	编译所需文件（附加）
<code>NO_PARSE</code>	创建只扫描的编译器	<code>globals.h</code> , <code>main.c</code> , <code>util.h</code> , <code>util.c</code> , <code>scan.h</code> , <code>scan.c</code>
<code>NO_ANALYZE</code>	创建只分析和扫描的编译器	<code>parse.h</code> , <code>parse.c</code>
<code>NO_CODE</code>	创建执行语义分析，但不生成代码的编译器	<code>syntab.h</code> , <code>syntab.c</code> , <code>analyze.h</code> , <code>analyze.c</code>

尽管这个TINY编译器设计得有些不太实际，但却有单个文件与阶段基本一致的好处，在以后的章节中将会一个一个地学到这些文件。

任何一个ANSI C编译器都可编译TINY编译器。假定可执行文件是`tiny`，通过使用以下命令：

```
tiny sample. tny
```

就可用它编译文本文件`sample.tny`中的TINY源程序。（如果省略了`.tny`，则编译器会自己添加`.tny`后缀）。屏幕上将会出现一个程序列表（它可被重定向到一个文件之上）并且（如当代码生成是被激活的）生成目标代码文件`sample.tm`（在下面将谈到的TM机中使用）。

在编辑列表的信息中有若干选项，以下的标志均可用：

标 志	设 置 效 果
EchoSource	将TINY源程序回显到带有行号的列表
TraceScan	当扫描程序识别出记号时，就显示每个记号的信息
TraceParse	将语法树以线性化格式显示
TraceAnalyze	显示符号表和类型检查的小结信息
TraceCode	打印有关代码文件的代码生成跟踪注释

1.7.3 TM 机

我们用该机器的汇编语言作为 TINY 编译器的目标语言。TM 机的指令仅够作为诸如 TINY 这样的小型语言的目标。实际上，TM 具有精减指令集计算机（RISC）的一些特性。在 RISC 中，所有的算法和测试均须在寄存器中进行，而且地址模式极为有限。为了使读者了解到该机的简便之处，我们将下面 C 表达式的代码

```
a[index] = 6
```

翻译成 TM 汇编语言（请读者将它与 1.3 节中相同语句假定的汇编语言比较一下）：

```
LDC 1, 0 ( 0 ) load 0 into reg 1
* 下面指令
* 假设 index 在存储器地址 10 中
LD 0, 10 ( 1 ) load val at (10+R1 into R0
LDC 1, 2 ( 0 ) load 2 into reg 1
MUL 0, 1, 0 put R1 * R0 into R0
LDC 1, 0 ( 0 ) load 0 into reg 1
* 下面指令
* 假设 a 在存储器地址 20 中
LDA 1, 20 ( 1 ) load 20 + R1 into R0
ADD 0, 1, 0 put R1 + R0 into R0
LDC 1, 6 ( 0 ) load 6 into reg 1
ST 1, 0 ( 0 ) store R1 at 0 + R0
```

我们注意到装入操作中有 3 个地址模式并且是由不同的指令给出的：LDC 是“装入常量”，LD 是“由存储器装入”，而 LDA 是“装入地址”。另外，该地址通常必须给成“寄存器 + 偏差”值。例如“10(1)”（上面代码的第 2 条指令），它代表在将偏差 10 加到寄存器 1 的内容中计算该地址。（因为在前面的指令中，0 已被装入到寄存器 1 中，这实际是指绝对位置 10）^①。我们还看到算术指令 MUL 和 ADD 可以是“三元”指令且只有寄存器操作数，其中可以单独确定结果的目标寄存器（1.3 节中的代码与此相反，其操作是“二元的”）。

TM 机的模拟程序直接从一个文件中读取汇编代码并执行它，因此应避免将由汇编语言翻译为机器代码的过程复杂化。但是，这个模拟程序并非是一个真正的汇编程序，它没有符号地址或标号。因此，TINY 编译器必须仍然计算跳转的绝对地址。此外为了避免与外部的输入/输出例程连接的复杂性，TM 机有内部整型的 I/O 设备；在模拟时，它们都对标准设备读写。

^① LDC 命令也要求一个“寄存器 + 偏差”的格式；但由于 TM 汇编程序格式简单统一，也就忽略了寄存器，偏差本身也作为常量装入。

通过使用任何一个ANSI C编译器，都可将tm.c 源代码编译成TM模拟程序。假定它的可执行文件叫作tm，通过发出命令

```
tm sample.tm
```

就可使用它了。其中，sample.tm是TIMY编译器由sample.tny源文件生成的代码文件。该命令引起代码文件的汇编和装入，接着就可交互地运行 TM模拟程序了。例如：如果sample.tny是程序清单1-1中的范例程序，则按以下计算就可得到7的阶乘：

```
tm sample. tm
TM simulation (enter h for help) ...
Enter command: go
Enter value for IN instruction: 7
OUT instruction prints: 5040
HALT: 0, 0, 0
Halted
Enter command: quit
Simulation done.
```

1.8 C-Minus：编译器项目的一种语言

附录A将描述一个比TINY更大的适用于编译器项目的语言。由于它受到C子集的严格限制，因此就称作C-Minus。它包括整型变量、整型数组以及函数（包含过程或空函数）；它还有局部、全局（静态）说明和（简单）递归函数，此外还有if语句和while语句；除此之外几乎什么也没有了。程序由函数序列和变量说明序列组成。最后必须是一个main函数，执行则由一个对main的调用开始^①。

程序清单1-2是在C-Minus中的一个程序示例，在其中通过递归函数编写了程序清单1-1中的阶乘程序。该程序的输入/输出由一个read函数提供，此外还有一个根据标准的C函数scanf和printf定义的write函数。

程序清单1-2 一个以其输入的阶乘为输出的C-Minus程序

```
int fact( int x )
/* recursive factorial function */
{ if (x > 1)
    return x * fact(x-1);
  else
    return 1;
}

void main( void )
{ int x;
  x = read();
  if (x > 0) write( fact(x) );
}
```

C-Minus是一个比TINY复杂的语言，在它的代码生成的要求中尤其如此；但是，TM机仍是其编译器合理的目标机器。附录A中有关于如何修改和扩充TINY编译器到C-Minus的内容。

① 为了与C-Minus的其他函数一致，main被说明为一个带有void参数列表的void函数。尽管这与ANSI C不同，但编译器还是接受了这种表示。

练习

1.1 从开发环境中找一个熟悉的编译器，并列出所有的相关程序，这些相关程序适用于与编译器一同进行操作，该编译器带有对其函数的简要描述。

1.2 下面是C中的赋值

```
a[i+1]=a[i]+2
```

利用1.3节中的类似例子为参考，画出这个表达式的分析树和语法树。

1.3 编译错误大致可分为两类：语法错误和语义错误。语法错误包括丢失记号和记号放置错误，例如算术表达式（ $2+3$ ，就缺少了右括号）。语义错误包括表达式中错误的类型及未说明的变量（在大多数语言中），例如赋值 $x=2$ ，其中 x 是一个数组变量。

a. 在你所选语言的每一个类型的错误中，再举出两个例子。

b. 找出一个你所熟悉的编译器，并判断它是在语义错误之前列出了所有的语法错误还是混合了语法和语义错误。这和编译的遍数有什么关系？

1.4 这个问题假设你有一个编译器，它有一个产生汇编语言输出的选项。

a. 判断你的编译器是否完成常量的合并优化。

b. 一个相关的但更先进的优化是常量传送的优化：当变量在表达式中有一个常量值时，它就由该值替代。例如，代码（在C语法中）：

```
x = 4;
```

```
y = x + 2;
```

通过常量传送（和常量合并）就变成代码：

```
x = 4;
```

```
y = 6;
```

判断你的编译器是否进行了常量的传送。

c. 为什么常量传送比常量合并更难，请说出尽可能多的理由。

d. 与常量传送和常量合并相关的情况是程序中被命名的常量的用法。利用命名的常量 x 代替一个变量，就可将上例翻译为以下的代码：

```
const int x = 4;
```

```
...
```

```
y = x + 2;
```

```
...
```

判断你的编译器是否在这些情况下执行传送/合并，这与b有何不同？

1.5 若你的编译器由键盘直接接受输入，则判断编译器是在生成出错信息之前读取整个程序还是在遇到它们时生成出错信息。这和编译的遍数有什么关系？

1.6 描述由以下程序完成的任务，并解释它们怎样与编译器相似或相关：

a. 语言预处理器

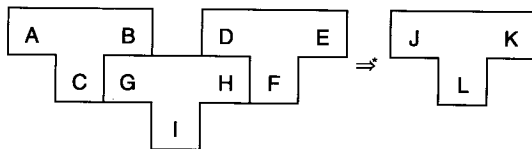
b. 格式打印程序

c. 文本格式化程序

1.7 假设有一个用C编写的由Pascal到C的翻译程序以及一个可运行的C编译器。请利用T型图来描述创建可运行的Pascal编译器的步骤。

1.8 我们已用箭头符 \Rightarrow 表示出将一个有两个T型图的格式变成只有一个T型图的格式。可将这个箭头符认为是一个“归约相关”，并构成它的传递闭包 \Rightarrow^* ，在其中允许有归约序列发生。下面给出了一个T型图，其中字母代表任意语言。请判断哪些语言必须

相等才能使归约有效，并且显示使它有效的单个归约步骤：



请给出一个该图所描述归约的实例。

- 1.9 在1.6节的图1-3中移植编译器的另一种方法是：对编译器生成的中间代码使用解释程序，并完全去掉后端。Pascal P-system使用的就是这种方法，Pascal P-system包括了生成P-代码的Pascal编译器、“一般的”栈式机器的汇编代码，以及模拟执行P-代码的P-代码解释程序。Pascal编译器和P-代码解释程序均用P-代码编写。
 - a. 假设有一个Pascal P-system，请描述在任一机器上得到可运行的Pascal编译器的步骤。
 - b. 描述在(a)中从你的系统得到一个可运行的本机代码编译器必需的步骤（也就是，为主机生成可执行代码的编译器，但不使用P-代码解释程序）。
- 1.10 可以认为移植编译器的过程有两个不同的操作：重置目标（retargeting）（为新机器修改编译器以生成目标代码）和重置主机（rehosting）（修改编译程序以在新机器上运行）。请根据T型图讨论两个操作的不同之处。

注意与参考

本章所讲到的大部分问题都将在以后各章中更加详细地再次提到，并且在以后的“注意与参考”中也会给出恰当的参考。例如，第2章会谈谈到Lex，第5章中则是Yacc，第6章中是类型检查、符号表和属性分析，第8章中则为代码生成、三元式代码和P-代码，第4章和第5章会研究错误处理。

Aho [1986]是编译器的一个标准的综合参考，尤其是在理论和算法方面。Fischer and LeBlanc [1991]中有许多有用的实行提示。在Fraser and Hanson [1995]和Holub [1990]中可找到对C编译器的完整描述。Gnu编译器是一个源代码在Internet上广泛应用的流行的C/C++编译器，Stallman [1994]详细描述了它。

如要了解程序设计语言的概念以及其与翻译程序相互影响的资料，可参考Louden [1993]或Sethi [1996]。

Hopcroft and Ullman [1979]对来自数学观点（不同于此处的实用观点）的自动机理论很有用。在这里还可以找到更多的有关乔姆斯基分类的内容（第3章中也会提到）。

Backus [1957]和Backus [1981]中有对早期FORTRAN编译器的描述。在Randell and Russell [1964]中有对早期的Algo160编译器的描述。Barron [1981]描述了Pascal编译器，在这里还提到了Pascal P-system（Nori [1981]）。

Kernighan [1975]中有1.2节中提到的Ratfor预处理器，Bratman [1961]介绍了1.6节的T型图。

本书着重于对大多数语言的翻译中有用的标准翻译技术。要对在基于Algol命令语言的主要传统语言之外的语言进行有效的翻译可能还需要其他技术。特别地，用于诸如ML和Haskell的函数语言翻译已经发展了许多新技术，其中一些可能会成为将来重要的通用技术。Appel [1992]、Peyton Jones [1992]和Peyton Jones [1987]均提到了这些技术。Peyton Jones [1987]还描述了Hindley-Milner类型检查（1.1节中已讲到过）。

第2章 词法分析

本章要点

- 扫描处理
- 正则表达式
- 有穷自动机
- 从正则表达式到 DFA
- TINY 扫描程序的实现
- 利用 Lex 自动生成扫描程序

编译器的扫描或词法分析 (lexical analysis) 阶段可将源程序读作字符文件并将其分为若干个记号。记号与自然语言中的单词类似：每一个记号都是表示源程序中信息单元的字符序列。典型的有：关键字 (keyword)，例如 `if` 和 `while`，它们是字母的固定串；标识符 (identifier) 是由用户定义的串，它们通常由字母和数字组成并由一个字母开头；特殊符号 (special symbol) 如算术符号 `+` 和 `*`、一些多字符符号，如 `>=` 和 `<>`。在各种情况中，记号都表示由扫描程序从剩余的输入字符的开头识别或匹配的某种字符格式。

由于扫描程序的任务是格式匹配的一种特殊情况，所以需要研究在扫描过程中的格式说明和识别方法，其中最主要的是正则表达式 (regular expression) 和有穷自动机 (finite automata)。但是，扫描程序也是处理源代码输入的编译器部分，而且由于这个输入经常需要非常多的额外时间，扫描程序的操作也就必须尽可能地高效了。因此还需十分注意扫描程序结构的实际细节。

扫描程序问题的研究可分为以下几个部分：首先，给出扫描程序操作的一个概貌以及所涉及到的结构和概念。其次是学习正则表达式，它是用于表示构成程序设计语言的词法结构的串格式的标准表示法。接着是有穷状态机器或称有穷自动机，它是对由正则表达式给出的串格式的识别算法。此外还研究用正则表达式构造有穷自动机的过程。之后再讨论当有穷自动机表示识别过程时，如何实际编写执行该过程的程序。另外还有 TINY 语言扫描程序的一个完整的实现过程。最后再看到自动产生扫描器生成器的过程和方法，并使用 Lex 再次实现 TINY 的扫描程序，它是适用于 Unix 和其他系统的标准扫描生成器。

2.1 扫描处理

扫描程序的任务是从源代码中读取字符并形成由编译器的以后部分 (通常是分析程序) 处理的逻辑单元。由扫描程序生成的逻辑单元称作记号 (token)，将字符组合成记号与在一个英语句子中将字母构成单词并确定单词的含义很相像。此时它的任务很像拼写。

记号通常定义为枚举类型的逻辑项。例如，记号在 C 中可被定义为[⊙]：

⊙ L 在一种没有列举类型的语言中，则只能将记号直接定义为符号数值。因此在老版本的 C 中有时可看到：

```
#define IF 256
#define THEN 257
#define ELSE 258
...
```

(这些数之所以是从 256 开始是为了避免与 ASCII 的数值混淆。)

```
typedef enum
{ IF, THEN, ELSEPLUS, MINUS, NUM, ID, ... }
TokenType;
```

记号有若干种类型，这其中包括了保留字（reserved word），如IF和THEN，它们表示字符串“if”和“then”；第2类是特殊符号（special symbol），如算术符号加（PLUS）和减（MINUS），它们表示字符“+”和“-”。第3类是表示多字符串的记号，如NUM和ID，它们分别表示数字和标识符。

作为逻辑项的记号必须与它们所表示的字符串完全区分开来。例如：保留字记号IF须与它表示的两个字符“if”的串相区别。为了使这个区别更明显，由记号表示的字符串有时称作它的串值（string value）或它的词义（lexeme）。某些记号只有一个词义：保留字就具有这个特性。但记号还可能表示无限多个语义。例如标识符全由单个记号ID表示，然而标识符有许多不同的串值来表示它们的单个名字。因为编译器必须掌握它们在符号表中的情况，所以不能忽略这些名字。因此，扫描程序也需用至少一些记号来构造串值。任何与记号相关的值都是记号的属性（attribute），而串值就是属性的示例。记号还可有其他的属性。例如，NUM记号可有一个诸如“32767”的串值属性，它是由5个数字字符组成，但它还会有一个由其值计算所得的真实值32767组成的数字值属性。在诸如PLUS这样的特殊符号记号中，不仅有串值“+”还有与之相关的真实算术操作+。实际上，记号符号本身就可看作是简单的其他属性，而记号就是它所有属性的总和。

为了后面的处理，扫描程序要求至少具有与记号所需相等的属性。例如要计算NUM记号的串值，但由于从它的串值就可计算，因此也就无需立刻计算它的数字值了。另一方面，如果计算它的数字值，就会丢掉它的串值。有时扫描程序本身会完成在恰当位置记录属性所需的操作，或直接将属性传到编译器后面的阶段。例如，扫描程序能利用标识符的串值将其输入到符号表中，或在后面传送它。

由于扫描程序必须计算每一个记号的若干属性，所以将所有的属性收集到一个单独构造的数据类型中是很有用的，这种数据类型称作记号记录（token record）。可在C中将这样的记录说明为：

```
typedef struct
{ TokenType tokenval;
  char * stringval;
  int numval;
} TokenRecord;
```

或可能作为一个联合

```
typedef struct
{ TokenType tokenval;
  union
  { char * stringval;
    int numval;
  } attribute;
} TokenRecord;
```

（以上假设只有标识符需要串值属性，只有数字需要数值属性）。对于扫描程序一个更普通的安排是只返回记号值并将变量的其他属性放在编译器的其他部分访问得到的地方。

尽管扫描程序的任务是将整个源程序转换为记号序列，但扫描程序却很少一次性地完成它。实际上，扫描程序是在分析程序的控制下进行操作的，它通过函数从输入中返回有关命令的下

一个记号，该函数有与C说明相似的说明：

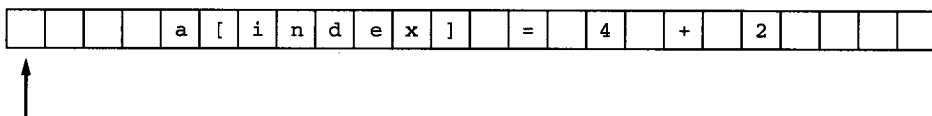
```
TokenType getToken( void );
```

这个方式中声明的`getToken`函数在调用时从输入中返回下一个记号，并计算诸如记号串值这样的附加属性。输入字符串通常并不给这个函数提供参数，但参数却被保存在缓冲区中或由系统输入设备提供。

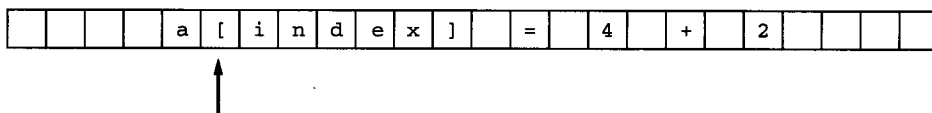
请考虑在`getToken`的操作示例中以下的C源代码行，在第1章中已用到过它：

```
a [ index ] = 4 + 2
```

假定将这个代码行放在一个输入缓冲区中，它的下一个输入字符由箭头指出，如下所示：



一个对`getToken`的调用现在需要跳过下面的4个空格，识别由单个字符`a`组成的串“`a`”，并返回记号值`ID`作为下个记号，此时的输入缓冲区如下所示：



因此，`getToken`随后的调用将再次从左括号字符开始识别过程。

现在开始研究在字符串中定义和识别格式的方法。

2.2 正则表达式

正则表达式表示字符串的格式。正则表达式 r 完全由它所匹配的串集来定义。这个集合称为由正则表达式生成的语言 (language generated by the regular expression)，写作 $L(r)$ 。此处的语言只表示“串的集合”，它与程序设计语言并无特殊关系（至少在此处是这样的）。该语言首先依赖于适用的字符集，它一般是 ASCII 字符的集合或它的某个子集。有时该集比 ASCII 字符的集合更普通一些，此处集合的元素称作符号 (symbol)。这个正规符号的集合称作字母表 (alphabet) 并且常写作希腊符号 Σ (sigma)。

正则表达式 r 还包括字母表中的字符，但这些字符具有不同的含义：在正则表达式中，所有的符号指的都是模式。在本章中，所有模式都是用黑体写出以与作为模式的字符区分开来。因此，**a** 就是一个作为模式的字符 `a`。

最后，正则表达式 r 还可包括有特殊含义的字符。这样的字符称作元字符 (metacharacter) 或元符号 (metasymbol)。它们并不是字母表中的正规字符，否则当其作为元字符时就与作为字母表中的字符时很难区分了。但是通常不可能要求这种排斥，而且也必须使用一个惯例来区分元字符的这两种用途。在很多情况下，它是通过使用可“关掉”元字符的特殊意义的转义字符 (escape character) 做到的。源代码层一般是反斜杠和引号。如果转义字符是字母表中的正规字符，则请注意它们本身也就是元字符了。

2.2.1 正则表达式的定义

现在通过讲解每个模式所生成的不同语言来描述正则表达式的含义。首先讲一下基本正则

表达式的集合（它是由单个符号组成），之后再描述从已有的正则表达式生成一个新的正则表达式的运算。它同构造算术表达式的方法类似：基本的算术表达式是由数字组成，如43和2.5。算术运算的加法和乘法等可用来从已有的表达式中产生新的表达式，如在 $43 * 2.5$ 和 $43 * 2.5 + 1.4$ 中。

从它们只包括了最基本的运算和元符号这一方面而言，这里所讲到的一组正则表达式都是最小的，以后还会讲得更深一些。

1) 基本正则表达式 它们是字母表中的单个字符且自身匹配。假设 a 是字母表中的任一字符，则指定正则表达式 a 通过书写 $L(a) = \{a\}$ 来匹配 a 字符。而特殊情况还要用到另外两个字符。有时需要指出空串（empty string）的匹配，空串就是不包含任何字符的串。空串用 ϵ (epsilon) 来表示，元符号 ϵ （黑体 ϵ ）是通过设定 $L(\epsilon) = \{\epsilon\}$ 来定义的。偶尔还需要写出一个与任何串都不匹配的符号，它的语言为空集（empty set），写作 $\{\}$ 。我们用符号 ϕ 来表示，并写作 $L(\phi) = \{\}$ 。请注意 $\{\}$ 和 $\{\epsilon\}$ 的区别： $\{\}$ 集不包括任何串，而 $\{\epsilon\}$ 则包含一个没有任何字符的串。

2) 正则表达式运算 在正则表达式中有3种基本运算：从各选择对象中选择，用元字符 $|$ （竖线）表示。连结，由并置表示（不用元字符）。重复或“闭包”，由元字符 $*$ 表示。后面通过为匹配了的串的语言提供相应的集合结构依次讨论以上3种基本运算。

3) 从各选择对象中选择 如果 r 和 s 是正则表达式，那么正则表达式 $r | s$ 可匹配被 r 或 s 匹配的任意串。从语言方面来看， $r | s$ 语言是 r 语言和 s 语言的联合（union），或 $L(r | s) = L(r) \cup L(s)$ 。以下是一个简单例子：正则表达式 $a | b$ 匹配了 a 或 b 中的任一字符，即 $L(a | b) = L(a) \cup L(b) = \{a\} \cup \{b\} = \{a, b\}$ 。又例如表达式 $a | \epsilon$ 匹配单个字符 a 或空串（不包括任何字符），也就是 $L(a | \epsilon) = \{a, \epsilon\}$ 。

还可在多个选择对象中选择，因此 $L(a | b | c | d) = \{a, b, c, d\}$ 也成立。有时还用点号表示选择的一个长序列，如 $a | b | \dots | z$ ，它表示匹配 $a \sim z$ 的任何小写字母。

4) 连结 正则表达式 r 和正则表达式 s 的连结可写作 rs ，它匹配两串连结的任何一个串，其中第1个匹配 r ，第2个匹配 s 。例如：正则表达式 ab 只匹配 ab ，而正则表达式 $(a | b)c$ 则匹配串 ac 和 bc （下面将简要介绍括号在这个正则表达式中作为元字符的作用）。

可通过由定义串的两个集合的连结所生成的语言来讲解连结的功能。假设有串 S_1 和 S_2 的两个集合，串 $S_1 S_2$ 的连结集合是将串 S_2 完全附加到串 S_1 上的集合。例如若 $S_1 = \{aa, b\}$ ， $S_2 = \{a, bb\}$ ，则 $S_1 S_2 = \{aaa, aabb, ba, bbb\}$ 。现在可将正则表达式的连结运算定义如下： $L(rs) = L(r) L(s)$ ，因此（利用前面的示例）， $L((a | b)c) = L(a | b) L(c) = \{a, b\} \{c\} = \{ac, bc\}$ 。

连结还可扩展到两个以上的正则表达式： $L(r_1 r_2 \dots r_n) = L(r_1) L(r_2) \dots L(r_n)$ = 由将每一个 $L(r_1), \dots, L(r_n)$ 串连结而成的串集合。

5) 重复 正则表达式的重复有时称为Kleene闭包（Kleene closure），写作 r^* ，其中 r 是一个正则表达式。正则表达式 r^* 匹配串的任意有穷连结，每个连结均匹配 r 。例如 a^* 匹配串 ϵ 、 a 、 aa 、 $aaa \dots$ （它与 ϵ 匹配是因为 ϵ 是与 a 匹配的非串连结）。通过为串集合定义一个相似运算 $*$ ，就可用生成的语言定义重复运算了。假设有一个串的集合 S ，则

$$S^* = \{\epsilon\} \cup S \cup SS \cup SSS \cup \dots$$

这是一个无穷集的联合，但是其中的每一个元素都是 S 中串的有穷连结。有时集合 S^* 可写作：

$$S^* = \bigcup_{n=0}^{\infty} S^n$$

其中 $S^n = S \dots S$ ，它是 S 的 n 次连结（ $S^0 = \{\epsilon\}$ ）。

现在可如下定义正则表达式的重复运算：

$$L(r^*) = L(r)^*$$

例：在正则表达式 $(a|bb)^*$ (括号作为元字符的原因将在后面解释) 中，该正则表达式与以下串任意匹配： ε 、 a 、 bb 、 aa 、 abb 、 bba 、 $bbbb$ 、 aaa 、 $aabb$ 等等。在语言方面， $L((a|bb)^*) = L(a|bb)^* = \{a, bb\}^* = \{\varepsilon, a, bb, aa, abb, bba, bbbb, aaa, aabb, abba, abbbb, bbaa, \dots\}$ 。

6) 运算的优先和括号的使用 前面的内容忽略了选择、连结和重复运算的优先问题。例如对于正则表达式 $a|b^*$ ，是将它解释为 $(a|b)^*$ 还是 $a|(b^*)$ 呢 (这里有一个很大的差别，因为 $L((a|b)^*) = \{\varepsilon, a, b, aa, ab, ba, bb, \dots\}$ ，但 $L(a|(b^*)) = \{\varepsilon, a, b, bb, bbb, \dots\}$)？标准惯例是重复的优先权较高，所以第2个解释是正确的。实际上，在这3个运算中， $*$ 优先权最高，连结其次， $|$ 最末。因此， $a|bc^*$ 就可解释为 $a|(b(c^*))$ ，而 $ab|c^*d$ 却解释为 $(ab)|((c^*)d)$ 。

当需指出与上述不同的优先顺序时，就必须使用括号。这就是为什么用 $(a|b)c$ 能表示选择比连结有更高的优先权的原因。而 $a|bc$ 则被解释为与 a 或 bc 匹配。类似地，没有括号的 $(a|bb)^*$ 应解释为 $a|bb^*$ ，它匹配 a 、 b 、 bb 、 bbb ...。此处括号的用法与其在算术中类似： $(3+4)*5=35$ ，而 $3+4*5=23$ ，这是因为 $*$ 的优先权比 $+$ 的高。

7) 正则表达式的名字 为较长的正则表达式提供一个简化了的名字很有用处，这样就不再需要在每次使用正则表达式时书写长长的表达式本身了。例如：如要为一个或多个数字序列写一个正则表达式，则可写作：

$(0|1|2|\dots|9)(0|1|2|\dots|9)^*$

或写作

*digit digit**

其中

digit = $0|1|2|\dots|9$

就是名字 *digit* 的正则定义 (regular definition) 了。

正则定义的使用带来了巨大的方便，但是它却增加了复杂性，它的名字本身也变成一个元符号，而且必须要找到一个方法能将这个名字与它的字符连结相区分开。在我们的例子中是用斜体来表示它的名字。请注意，在名字的定义中不能使用名字 (也就是递归地)——必须能够用它代表的正则表达式替换它们。

在考虑用一些例子来详细描述正则表达式的定义之前，先将有关正则表达式定义的所有信息收集在一起。

定义 正则表达式 (regular expression) 是以下的一种：

1. 基本 (basic) 正则表达式由一个单字符 a (其中 a 在正规字符的字母表中)，以及元字符 ε 或元字符 ϕ 组成。在第1种情况下， $L(a) = \{a\}$ ；在第2种情况下， $L(\varepsilon) = \{\varepsilon\}$ ；在第3种情况下， $L(\phi) = \{\}$ 。
2. $r|s$ 格式的表达式：其中 r 和 s 均是正则表达式。在这种情况下， $L(r|s) = L(r) \cup L(s)$ 。
3. rs 格式的表达式：其中 r 和 s 是正则表达式。在这种情况下， $L(rs) = L(r)L(s)$ 。
4. r^* 格式的表达式：其中 r 是正则表达式。在这种情况下， $L(r^*) = L(r)^*$ 。
5. (r) 格式的表达式：其中 r 是正则表达式。在这种情况下， $L((r)) = L(r)$ ，因此，括号并不改变语言，它们只调整运算的优先权。

我们注意到在上面这个定义中，(2)、(3) 和 (4) 的优先权与它们所列的顺序相反，也就

是：| 的优先权最低，连结次之，* 则最高。另外还注意到在这个定义中，6个符号—— ϕ 、 ε 、|、*、(和) 都有了元字符的含义。

本节后面将给出一些示例来详述上述定义，但它们并不经常作为记号描述在程序设计语言中出现。2.2.3节将讨论一些经常作为记号在程序设计语言中出现的常用正则表达式。

在下面的示例中，被匹配的串通常是英语描述，其任务是将描述翻译为正则表达式。包含了记号描述的语言手册是编译器的程序员最常见的。偶尔还需要变一下，也就是将正则表达式翻译为英语描述，我们也有一些此类的练习。

例2.1 在仅由字母表中的3个字符组成的简单字母表 $= \{a, b, c\}$ 中，考虑在这个字母表上的仅包括一个 b 的所有串的集合，这个集合由正则表达式

$$(a|c)^*b(a|c)^*$$

产生。尽管 b 出现在正则表达式的正中间，但字母 b 却无需位于被匹配的串的正中间。实际上，在 b 之前或之后的 a 或 c 的重复会发生不同的次数。因此，串 b 、 abc 、 $abaca$ 、 $baaaac$ 、 $ccbaca$ 和 $ccccccb$ 都与上面的正则表达式匹配。

例2.2 在与上面相同的字母表中，如果集合是包括了最多一个 b 的所有串，则这个集合的正则表达式可通过将上例的解作为一个解（与那些仅为一个 b 的串匹配），而正则表达式 $(a|c)^*$ 则作为另一个解（与 b 根本不匹配）来获取。因此有以下解：

$$(a|c)^*|(a|c)^*b(a|c)^*$$

下面是一个既允许 b 又允许空串在重复的 a 或 c 之间出现的另一个解：

$$(a|c)^*(b|\varepsilon)(a|c)^*$$

本例引出了正则表达式的一个重要问题：不同的正则表达式可生成相同的语言。虽然在实际中从未尝试着证实已找到了“最简单的”，例如最短的，正则表达式，但通常总是试图用尽可能简单的正则表达式来描述串的集合。这里有两个原因：首先在现实中极少有标准的“最短”解；其次，在研究用作识别正则表达式的方法时，那儿的算法无需首先将正则表达式简化就可将识别过程简化了。

例2.3 在字母表 $= \{a, b\}$ 上的串 S 的集合是由一个 b 及在其前后有相同数目的 a 组成：

$$S = \{b, aba, aabaa, aaabaaa, \dots\} = \{a^n b a^n | n \geq 0\}$$

正则表达式并不能描述这个集合，其原因在于重复运算只有闭包运算 $*$ 一种，它允许有任意次的重复。因此如要写出表达式 a^*ba^* （尽可能接近地得到 S 的正则表达式），就不能保证在 b 前后的 a 的数量是否相等了，它通常表示为“不能计算的正则表达式”。但若给出一个数学论证，则需使用有关正则表达式的称作 Pumping 引理（Pumping lemma）的著名定理，这将在自动机理论中学到，现在就不谈了。

很明显，并非用简单术语描述的所有串都可由正则表达式产生，因此为了与其他集合相区分，作为正则表达式语言的串集合称作正则集合（regular set）。非正则集合偶尔也作为串出现在需要由扫描程序识别的程序设计语言中，通常是在出现时才处理它们，我们也将它们放在扫描程序一节中讨论。

例2.4 在字母表 $= \{a, b, c\}$ 上的串 S 中，任意两个 b 都不相连，所以在任何两个 b 之间都至少有一个 a 或 c 。可分几步来构造这个正则表达式，首先是在每一个 b 后都有一个 a 或 c ，它写作：

$$(b(a|c))^*$$

将其与表达式 $(a|c)^*$ 合并, $(a|c)^*$ 是与完全不包含 b 的串匹配, 则写作:

$$((a|c)^*|(b(a|c))^*)^*$$

或考虑到 $(r^*|s^*)^*$ 与 $(r|s)^*$ 所匹配的串相同, 则:

$$((a|c)|(b(a|c)))^*$$

或

$$(a|c|ba|bc)^*$$

(警告! 这还不是正确答案)。

这个正则表达式产生的语言实际上具有了所需的特性, 即: 没有两个相连的 b (但这还不正确)。有时需要证明一下上面的这个说法, 也就是证明在 $L((a|c|ba|bc)^*)$ 中的所有串都不包括两个相连的 b 。该证明是通过串长度 (即串中字符数) 的归纳实现的。很显然, 它对于所有长度为 0、1 和 2 的串是正确的: 这些串实际是串 ε 、 a 、 c 、 aa 、 ac 、 ca 、 cc 、 ba 和 bc 。现在假设对于在长度 $i < n$ 的语言中的所有串也为真, 并令 s 是长度为 $n > 2$ 的语言中的一个串, 那么 s 包含了至少一个上面所列的非 ε 的串, 所以 $s = s_1s_2$, 其中 s_1 和 s_2 也是语言中的非 ε 串。通过假设归纳, 证明了 s_1 和 s_2 都没有两个相连的 b 。因此要使 s 本身包括两个相连的 b 的唯一方法是使 s_1 以一个 b 结尾, 而 s_2 以一个 b 开头。但又因为语言中的串都不以 b 结尾, 所以这是不可能的。

论证中的最后一个事实, 即由上面的正则表达式所产生的串都不以 b 结尾, 也显示了我们的解还不太正确: 它不产生串 b 、 ab 和 cb , 这 3 个都不包括两个相连的 b 。可以通过为其添加一个可选的结尾 b 来修改它, 如下所示:

$$(a|c|ba|bc)^*(b|\varepsilon)$$

这个正则表达式的镜像也生成了指定的语言:

$$(b|\varepsilon)(a|c|ab|cb)^*$$

以下也可生成相同的语言:

$$(notb|b\ notb)^*(b|\varepsilon)$$

其中 $notb = a|c$ 。这是一个使用了下标表达式名字的例子。由于无需将原表达式变得更复杂就可使 $notb$ 的定义调整为包括了除 b 以外的所有字符, 因此实际是在字母表较大时使用这个解。

例 2.5 本例给出了一个正则表达式, 要求用英语简要地描述它生成的语言。若有字母表 $= \{a, b, c\}$, 则正则表达式:

$$((b|c)^*a(b|c)^*a)^*(b|c)^*$$

生成了所有包括偶数个 a 的串的语言。为了看清它, 可考虑外层左重复之中的表达式:

$$(b|c)^*a(b|c)^*a$$

它生成的串是以 a 结尾且包含了两个 a (在这两个 a 之前或之间可有任意个 b 和 c)。重复这些串则得到所有以 a 结尾的串, 且 a 的个数是 2 的倍数 (即偶数)。在最后附加重复 $(b|c)^*$ (如前例所示) 则得到所需结果。

这个正则表达式还可写作:

$$(nota^*a\ nota^*a)^*nota^*$$

2.2.2 正则表达式的扩展

前面已给出了正则表达式的一个定义, 这个正则表达式使用了在所有应用程序中都常见到

运算的最小集合，而且使上面的示例仅限于使用3种基本运算（包括括号）。但是从以上这些示例中可看出仅利用这些运算符来编写正则表达式有时显得很笨拙，如果可用一个更有表达力的运算集合，那么创建的正则表达式就会更复杂一些。例如，使任意字符的匹配具有一个表示法很有用（我们现在须在一个长长的解中列出字母表中的每个字符）。除此之外，拥有字符范围的正则表达式和除单个字符以外所有字符的正则表达式都十分有效。

下面几段文字将描述前面所讨论的标准正则表达式的一些扩展情况，以及与它及类似情况相对应的新元符号。在大多数情况下并不存在通用术语，所以使用的是与在扫描程序生成器Lex中用到的类似的表示法，本章后面将会讲到Lex。实际上，以后要谈到的很多情况都会在对Lex的讨论中再次提到。并非所有使用正则表达式的应用程序都包括这些运算，即使是这样，也要用到不同的表示法。

下面是新运算的列表。

(1) 一个或多个重复

假若有一个正则表达式 r ， r 的重复是通过使用标准的闭包运算来描述，并写作 r^* 。它允许 r 被重复0次或更多次。0次并非是最典型的情况，一次或多次才是，这就要求至少有一个串匹配 r ，但空串 ε 却不行。例如在自然数中需要有一个数字序列，且至少要出现一个数字。如要匹配二进制数，就写作 $(0|1)^*$ ，它同样也可匹配不是一个数的空串。当然也可写作

$(0|1)(0|1)^*$

但是这种情况只出现在用+代替*的这个相关的标准表示法被开发之前： r^+ 表明 r 的一个或多个重复。因此，前面的二进制数的正则表达式可写作：

$(0|1)^+$

(2) 任意字符

为字母表中的任意字符进行匹配需要一个通常状况：无需特别运算，它只要求字母表中的每个字符都列在一个解中。句号“.”表示任意字符匹配的典型元字符，它不要求真正将字母表写出来。利用这个元字符就可为所有包含了至少一个 b 的串写出一个正则表达式，如下所示：

$.^*b.^*$

(3) 字符范围

我们经常需要写出字符的范围，例如所有的小写字母或所有的数字。直到现在都是在用表示法 $a|b|\dots|z$ 来表示小写字母，用 $0|1|\dots|9$ 来表示数字。还可针对这种情况使用一个特殊表示法，但常见的表示法是利用方括号和一个连字符，如 $[a-z]$ 是指所有小写字母， $[0-9]$ 则指数字。这种表示法还可用作表示单个的解，因此 $a|b|c$ 可写成 $[abc]$ ，它还可用于多个范围，如 $[a-zA-Z]$ 代表所有的大小写字母。这种普遍表示法称为字符类（character class）。例如， $[A-Z]$ 是假设位于A和Z之间的字符B、C等（一个可能的假设）且必须只能是A和Z之间的大写字母（ASCII字符集也可）。但 $[A-z]$ 则与 $[A-Za-z]$ 中的字符不匹配，甚至与ASCII字符集中的字符也不匹配。

(4) 不在给定集合中的任意字符

正如前面所见的，能够使要匹配的字符集中不包括单个字符很有用，这点可由用元字符表示“非”或解集合的互补运算来做到。例如，在逻辑中表示“非”的标准字符是波形符“~”，那么表示字母表中非 a 字符的正则表达式就是 $\sim a$ 。非 a 、 b 及 c 表示为：

$\sim(a|b|c)$

在Lex中使用的表示法是在连结中使用插入符“^”和上面所提的字符类来表示互补。例如，

任何非 a 的字符可写作 $[^a]$ ，任何非 a 、 b 及 c 的字符则写作：

```
[^abc]
```

(5) 可选的子表达式

有关串的最后一个是常见的情况是在特定的串中包括既可能出现又可能不出现的可选部分。例如，数字前既可有一个诸如 $+$ 或 $-$ 的先行符也可以没有。这可用解来表示，同在正则定义中是一样的：

```
natural = [0-9]+
signedNatural = natural | + natural | - natural
```

但这会很快变得麻烦起来，现在引入问号元字符 $r?$ 来表示由 r 匹配的串是可选的（或显示 r 的0个或1个拷贝）。因此上面那个先行符号的例子可写成：

```
natural = [0-9]+
signedNatural = (+|-)?natural
```

2.2.3 程序设计语言记号的正则表达式

在众多不同的程序设计语言中，程序设计语言记号可分为若干个相当标准的有限种类。第1类是保留字的，有时称为关键字（keyword），它们是语言中具有特殊含意的字母表字符的固定串。例如：在Pascal、C和Ada语言中的`if`、`while`和`do`。另一个范围由特殊符号组成，它包括算术运算符、赋值和等式。它们可以是一个字符，如`=`，也可是多个字符如：`=`或`++`。第3种由标识符（identifier）组成，它们通常被定义为以字母开头的字母和数字序列。最后一种包括了文字（literal）或常量（constant），如数字常量42和3.14159，如串文字“hello, world, ”，及字符“ a ”和“ b ”。在这里仅讨论一下它们的典型正则表达式以及与记号识别相关的问题，本章后面还会更详细地谈到实际识别问题。

1) 数 数可以仅是数字（自然数）、十进制数、或带有指数的数（由 e 或 E 表示）的序列。例如：2.71E-2表示数.0271。可用正则定义将这些数表示如下：

```
nat = [0-9]+
signedNat = (+|-)?nat
number = signedNat("." nat) ? (E signedNat)?
```

此处，在引号中用了一个十进制的点来强调它应直接匹配且不可被解释为一个元字符。

2) 保留字和标识符 正则表达式中最简单的就是保留字了，它们由字符的固定序列表示。如果要所有的保留字收集到一个定义中，就可写成：

```
reserved = if | while | do | ...
```

相反地，标识符是不固定的字符串。通常，标识符必须由一个字母开头且只包含字母和数字。可用以下的正则定义表示：

```
letter = [a-zA-Z]
digit = [0-9]
identifier = letter(letter | digit)*
```

3) 注释 注释在扫描过程中一般是被忽略的^①。然而扫描程序必须识别注释并舍弃它们。因此尽管扫描程序可能没有清晰的常量记号（可将其称为“伪记号 pseudotoken”），仍需要给注释编写出正则表达式。注释可有若干个不同的格式。通常，它们可以是前后为分隔符的自由

① 它们有时可包括编译目录。

格式，例如：

```
{ this is a Pascal comment }
/* this is a C comment */
```

或由一个或多个特殊字符开头并直到该行的结尾，如在

```
; this is a Scheme comment
-- this is an Ada comment
```

中。

为有单个字符的分隔符的注释(如 Pascal 注释)编写正则表达式并不难，且为那些从行的特殊字符到行尾编写正则表达式也不难。例如 Pascal 注释可写作：

```
{(~)}*
```

其中，用 `~` 表示“非”，并假设字符 `}` 作为元字符没有意义（在 Lex 中的表示与之不同，本章后面将会提到）。类似地，一个 Ada 注释可被正则表达式

```
--(~\newline)*
```

匹配。其中，假设 `\newline` 匹配一行的末尾（在许多系统中可写作 `\n`），`-` 字符作为元字符没有意义，该行的结尾并未包括在注释本身之中（2.6 节将谈到如何在 Lex 中书写它）。

为同 C 注释一样，其中的分隔符如多于一个字符时，则编写正则表达式就要困难许多。例如串集合 `ba... (没有 ab) ... ab`（用 `ba... ab` 代替 C 的分隔符 `/*...*/`，这是因为星号，有时还有前斜杠要求特殊处理的元字符）。不能简单地写作：

```
ba(~(ab))*ab
```

由于“非”运算通常限制为只能是单个字符而不能使用字符串。可尝试用 `~a`、`~b` 和 `~(a|b)` 为 `~(ab)` 写出一个定义来，但这却没有多大用处。其中的一个解是：

```
b*(a*~(a|b)b*)*a*
```

然而这很难读取（且难证明是正确的）。因此，C 注释的正则表达式是如此之难以至于在实际中几乎从未编写过。实际上，这种情况在真正的扫描程序中经常是通过特殊办法解决的，本章后面将会提到它。

最后，在识别注释时会遇到的另一个复杂的问题是：在一些程序设计语言中，注释可被嵌套。例如 Modula-2 允许格式注释：

```
(* this is (*a Modula-2 *) comment *)
```

在这种嵌套注释中，注释分隔符必须成对出现，故以下注释在 Modula-2 中是不正确的：

```
(* this is ( * illegal in Modula-2 *)
```

注释的嵌套要求扫描程序计算分隔符的数量，但我们又注意到在例 2.3（2.2.1 节）中，正则表达式不能表示计数操作。实际上，一个简单的计算器配置可作为这个问题的特殊解（参见练习）。

4) 二义性、空白格和先行 在程序设计语言记号使用正则表达式的描述中，有一些串经常可被不同的正则表达式匹配。例如：诸如 `if` 和 `while` 的串可以既是标识符又可以是关键字。类似地，串 `<>` 可解释为表示两个记号（“小于号”和“大于号”）或单个符号（“不等于”）。程序设计语言定义必须规定出应遵守哪个解释，但正则表达式本身却无法做到它。相反地，语言定义必须给出无二义性规则（disambiguating rules），由其回答每一种情况下的含义。

下面给出处理示例的两个典型规则。首先当串既可以是标识符也可以是关键字时，则通常认为它是关键字。这暗示着使用术语保留字（reserved word），其含义只是关键字不能同时也

是标识符。其次，当串可以是单个记号也可以是若干记号的序列时，则通常解释为单个记号。这常常被称作最长子串原理（principle of longest substring）：可组成单个记号的字符的最长串在任何时候都是假设为代表下一个记号^①。

在使用最长子串原理时会出现记号分隔符（token delimiter）的问题，即表示那些在某时不能代表记号的长串的字符。分隔符应是肯定为其他记号一部分的字符。例如在串 `xtemp=ytemp` 中，等号将标识符 `xtemp` 分开，这是因为 `=` 不能作为标识符的部分出现。通常也认为空格、新行和制表位是记号分隔符：因此 `while x..` 就解释为包含了两个记号——保留字 `while` 和带有名字 `x` 的标识符，这是因为一个空格将两个字符串分开。在这种情况下，定义空白格伪记号非常有用，它与注释伪记号相类似，但注释伪记号仅仅是在扫描程序内部区分其他记号。实际上，注释本身也经常作为分隔符，因此例如 C 代码片段：

```
do / ** / if
```

表示的就是两个保留字 `do` 和 `if`，而不是标识符 `doif`。

程序设计语言中的空白格伪记号的典型定义是：

```
whitespace= (newline | blank | tab | comment)+
```

其中，等号右边的标识符代表恰当的字符或串。请注意：空白格通常不是作为记号分隔符，而是被忽略掉。指定这个行为的语言叫作自由格式语言（free format）。除自由格式语言之外还可以是一些诸如 FORTRAN 的固定格式语言，以及各种使用缩排格式的语言，例如越位规则（offside rule）（参见“注意与参考”一节）。自由格式语言的扫描程序必须在检查任意记号分隔功能之后舍弃掉空白格。

分隔符结束记号串，但它们并不是记号本身的一部分。因此，扫描程序必须处理先行（lookahead）问题：当它遇到一个分隔符时，它必须作出安排分隔符不会从输入的其他部分中删除，方法是将分隔符返回到输入串（“备份”）或在将字符从输入中删除之前先行。在大多数情况下，只有单个字符才需要这样做（“单个字符先行”）。例如在串 `xtemp=ytemp` 中，当遇到 `=` 时，就可找到标识符 `xtemp` 的结尾，且 `=` 必须保留在输入中，这是因为它表示要识别下一个记号。还应注意，在识别记号时可能不需要使用先行。例如，等号可能是以 `=` 开头的唯一字符，此时无需考虑下一个字符就可立即识别出它了。

有时语言可能要求不仅仅是单个字符先行，且扫描程序必须准备好可以备份任意多个字符。在这种情况下，输入字符的缓冲和为追踪而标出位置就给扫描程序的设计带来了问题（本章后面将会讨论到其中的一些问题）。

FORTRAN 是一个并不遵守上面所谈的诸多原则的典型语言。它是固定格式语言，它的空白格在翻译开始之前已由预处理器删除了。因此，下面的 FORTRAN 行

```
I F ( X 2 . EQ . 0 ) T H E N
```

在编译器中就是

```
IF(X2.EQ.0)THEN
```

所以空白格再也不是分隔符了。FORTRAN 中也没有保留字，故所有的关键字也可以是标识符，输入每行中字符串的位置对于确定将要识别的记号十分重要。例如，下面代码行在 FORTRAN 中是完全正确的：

```
IF( IF . EQ . 0 ) THEN THEN = 1.0
```

① 有时这也称作“最大咀嚼”定理。

第1个IF和THEN都是关键字，而第2个IF和THEN则是表示变量的标识符。这样的结果是FORTRAN的扫描程序必须能够追踪代码行中的任意位置。例如：

```
DO99I=1,10
```

它引起循环体为第99行代码的循环。在Pascal中，则表示为for i := 1 to 10另一方面，若将逗号改为句号：

```
DO99I=1.10
```

就将代码的含义完全改变了：它将值1.1赋给了名字为DO99I的变量。因此，扫描程序只有到它接触到逗号（句号）时才能得出起始的DO。在这种情况下，它可能只得追踪到行的开头并且由此开始。

2.3 有穷自动机

有穷自动机，或有穷状态的机器，是描述（或“机器”）特定类型算法的数学方法。特别地，有穷自动机可用作描述在输入串中识别模式的过程，因此也能用作构造扫描程序。当然有穷自动机与正则表达式之间有着很密切的关系，在下一节中大家将会看到如何从正则表达式中构造有穷自动机。但在学习有穷自动机之前，先看一个说明的示例。

通过下面的正则表达式可在程序设计语言中给出标识符模式的一般定义（假设已定义了letter和digit）：

```
identifier= letter ( letter | digit ) *
```

它代表以一个字母开头且其后为任意字母和 / 或数字序列的串。识别这样的一个串的过程可表示为图2-1。在此图中，标明数字1和2的圆圈表示的是状态（state），它们表示其中记录已被发现的模式的数量在识别过程中的位置。带有箭头的线代表由记录一个状态向另一个状态的转换（transition），该转换依赖于所标字符的匹配。在较简单的图示中，状态1是初始状态（start state）或识别过程开始的状态。

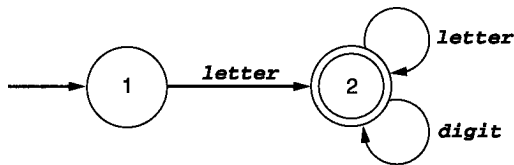


图2-1 标识符的有穷自动机

按照惯例，初始状态表示为一个“不来自任何地方”且指向它的未作标识的箭头线。状态2代表有一单个字母已被匹配的点（表示为从状态1到状态2的转换，且其上标有letter）。一旦位于状态2中，就可看到任何数量的字母和 / 或数字，它们的匹配又使我们回到了状态2。代表识别过程结束的状态称作接受状态（accepting state），当位于其中时就可说明成功了，在图中它表示为在状态的边界画出双线。它们可能不只一个。在上面的例图中，状态2就是一个接受状态，它表示在看到一字母之后，随后的任何字母和数字序列（也包括根本没有）都表示一个正规的标识符。

将真实字符串识别为标识符的过程现在可通过列出在识别过程中所用到的状态和转换的序列来表示。例如，将xtemp识别为标识符的过程可表示为：

```
1 x 2 t 2 e 2 m 2 p 2
```

在此图中，用在每一步中匹配的字母标出了每一个转换。

2.3.1 确定性有穷自动机的定义

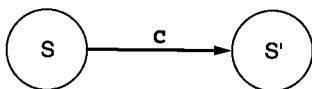
因为上面所示的例图很方便地展示出算法过程，所以它对于有穷自动机的描述很有用处。

但是偶尔还需使用有穷自动机的更正式的描述，现在就给出一个数学定义。但绝大多数情况并不需要这么抽象，且在大多数示例中也只使用示意图。有穷自动机还有其他的描述，尤其是表格，表格对于将算法转换成运行代码很有用途。在需要它的时候我们将会谈到它。

另外读者还需注意：我们一直在讨论的是确定性的（deterministic）有穷自动机，即：下一个状态由当前状态和当前输入字符唯一给出的自动机。非确定性的有穷自动机是由它产生的。本节稍后将谈到它。

定义：DFA（确定性有穷自动机） M 由字母表、状态集合 S 、转换函数 $T: S \times S \rightarrow S$ 、初始状态 $s_0 \in S$ 以及接受状态的集合 $A \subseteq S$ 组成。由 M 接受的且写作 $L(M)$ 被定义为字符 $c_1 c_2 \dots c_n$ 串的集合，其中每个 c_i ，存在状态 $s_1 = T(s_0, c_1)$, $s_2 = T(s_1, c_2)$, \dots , $s_n = T(s_{n-1}, c_n)$ ，其中 s_n 是 A （即一个接受状态）的一个元素。

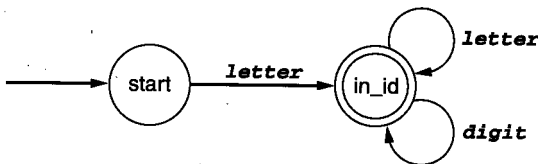
有关这个定义请注意以下几点。 $S \times S$ 指的是 S 和 S 的笛卡尔积或叉积：集合对 (s, c) ，其中 $s \in S$ 且 $c \in S$ 。如果有一个标为 c 的由状态 s 到状态 s' 的转换，则函数 T 记录转换： $T(s, c) = s'$ 。与 M 相应的示图片段如下：



当接受状态序列 $s_1 = T(s_0, c_1)$, $s_2 = T(s_1, c_2)$, \dots , $s_n = T(s_{n-1}, c_n)$ 存在，且 s_n 是一个接受状态时，它表示如下所示的意思：

$$\rightarrow s_0 \xrightarrow{c_1} s_1 \xrightarrow{c_2} s_2 \longrightarrow \dots \longrightarrow s_{n-1} \xrightarrow{c_n} s_n$$

在DFA的定义和标识符示例的图示之间有许多区别。首先，在标识符图中的状态使用了数字，而定义并未用数字对状态集合作出限制。实际上可以为状态使用任何标识系统，这其中也包括了名字。例如：下图与图2-1完全一样：



在这里就称作状态`start`（因为它是初始状态）和`in_id`（因为我们发现了一个字母并识别其后的任何字母和数字后面的标识符）。这个图示中的状态集合现在变成了 $\{\text{start}, \text{in_id}\}$ ，而不是 $\{1, 2\}$ 了。

图示与定义的第2个区别在于转换不是用字符标出而是表示为字符集合的名字。例如，名字`letter`表示根据以下正则定义的字母表中的任意字母：

`letter = [a - zA - Z]`

因为如要为每个小写字母和每个大写字母画出总共52个单独的转换非常麻烦，所以这是定义的一个便利的扩展。本章后面还会用到这个定义的扩展。

图示与定义的第3个区别更为重要：定义将转换表示为一个函数 $T: S \times S \rightarrow S$ 。这意味着 $T(s, c)$ 必须使每个 s 和 c 都有一个值。但在图中，只有当 c 是字母时，才定义 $T(\text{start}, c)$ ；而也只有当 c 是字母或数字时，才定义 $T(\text{in_id}, c)$ 。那么，所丢失的转换跑到哪里去了呢？答案是它们表示了出错——即在识别标识符时，我们不能接受除来自初始状态之外的任何字符以及

这之后的字母或数字^①。按照惯例，这些出错转换（error transition）在图中并没有画出来而只是假设它总是存在着。如果要画出它们，则标识符的图示应如图2-2所示：

在该图中，我们将新状态 *error* 标出来了（这是因为它表示出错的发生），而且还标出了出错转换 *other*。按照惯例，*other* 表示并未出现在来自于它所起源的状态的任何其他转换中的任意字符，因此 *other* 的定义来自于初始状态为：

other = \sim letter

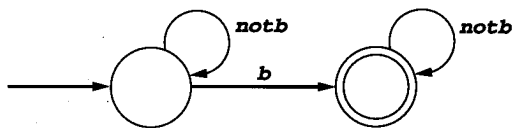
来自 *in_id* 状态的 *other* 的定义是：

other = \sim (letter|digit)

请注意，来自出错状态的所有转换都要回到其本身（这些转换用 *any* 标出以表示在这个转换中得出的任何字符）。另外，出错状态是非接受的，因此一旦发生一个出错，则无法从出错状态中逃避开来，而且再也不能接受串了。

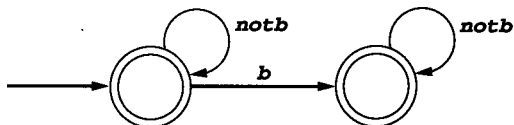
下面是DFA的一些示例，其中也有一些是在上一节中提到过的。

例2.6 串中仅有一个 *b* 的集合被下示的DFA接受：



请注意，在这里并未标出状态。当无需用名字来指出状态时就忽略标签。

例2.7 包含最多一个 *b* 的串的集合被下示的DFA接受：



请注意这个DFA是如何修改前例中的DFA，它将初始状态变成另一个的接受状态。

例2.8 前一节给出了科学表示法中数字常量的正则表达式，如下所示：

```
nat = [0-9]+
signedNat = (+|-)?nat
number = signedNat("." nat)? (E signedNat)?
```

我们想为由这些定义匹配的串写出DFA，但是先如下重写它会更为有用：

```
digit = [0-9]
nat = digit+
```

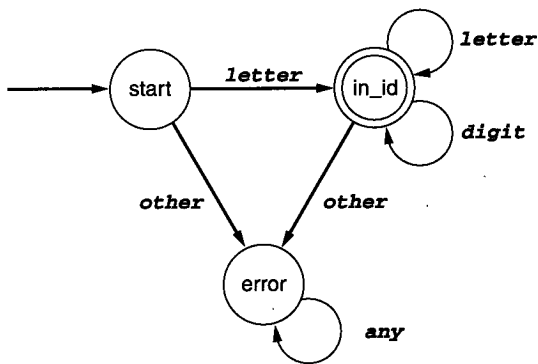


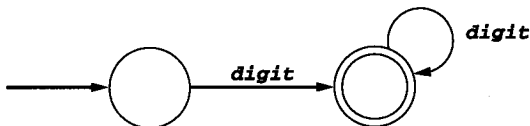
图2-2 带有出错转换的标识符的有穷自动机

^① 在实际情况下，这些非文字数字的字符意味着根本就没有标识符（如果是在初始状态中）或遇到了一个结束标识符的识别的分隔符（如果是在接受状态中）。本节后面将介绍如何处理这些情况。

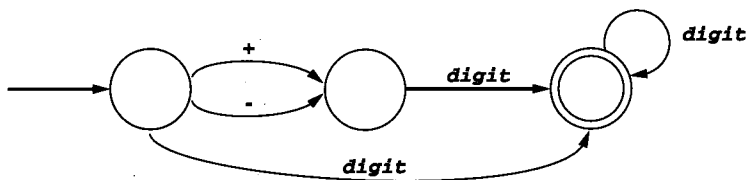
```
signedNat = (+|-)?nat
```

```
number = signedNat("." nat)? (E signedNat)?
```

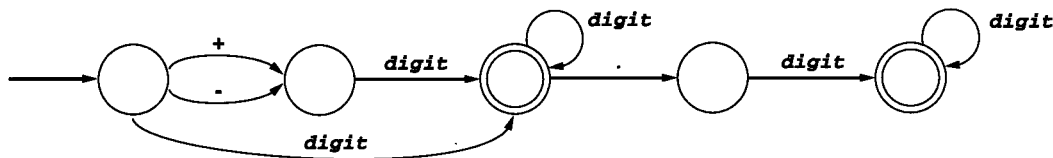
如下为`nat`写出一个DFA是非常简单（请记住 $a^+ = aa^*$ 对任意的 a 均成立）的：



由于可选标记，`signedNat`要略难一些，但是可注意到`signedNat`是以一个数字或一个标记与数字开头，并写作以下的DFA：



在它上面添加可选的小数部分也很简单，如下所示：



请注意，它有两个接受状态，它们表示小数部分是可选的。

最后需要添加可选的指数部分。要做到它，就要指出指数部分必须是以字母 `E` 开头，且只能发生在前面的某个接受状态之后，图2-3是最终的图。

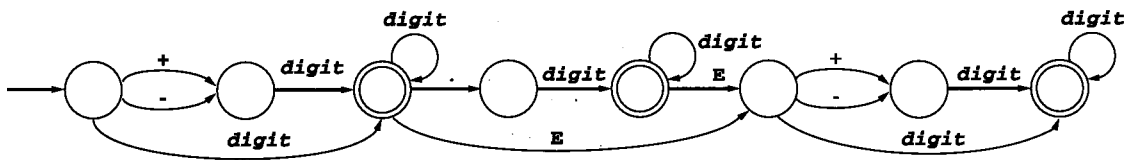
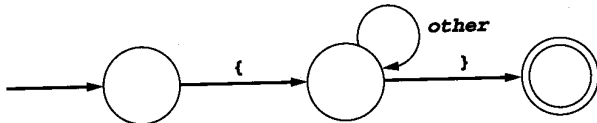


图2-3 浮点数的有穷自动机

例2.9 使用DFA可描述非嵌套注释。例如，前后有花括号的注释可由以下的DFA接受：



在这种情况下，`other`意味着除了右边花括号外的所有字符。这个DFA与第2.2.4节中所写的正则表达式 $\{(\sim\})^*\}$ 相对应。

我们注意到在2.2.4中，为被两个字符的序列分隔开的注释编写一个正则表达式很难，C注释的格式`/*... (no*/s) .../`就是这样的。编写接受这个注释的DFA比编写它的正则表达式实际上要简单许多，图2-4中的DFA就是这样的C注释。

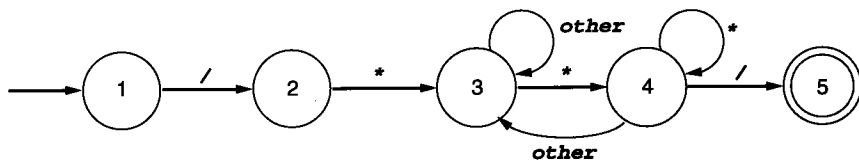


图2-4 有C风格注释的有穷自动机

在该图中，由状态3到其自身的 *other* 转换表示除 “*” 之外的所有字符，但由状态4到状态3的 *other* 转换则表示除 “*” 和 “/” 之外的所有字符。为了简便起见，还给状态编了号，但仍能为状态赋予更多有意义的名字，如下所示（在括号中带有其相应的编号）：start (1)、entering_comment (2)、in_comment (3)、exiting_comment (4) 和 finish (5)。

2.3.2 先行、回溯和非确定性自动机

作为根据模式接受字符串的表示算法的一种方法，我们已经学习了 DFA。正如同读者可能早已猜到的一样，模式的正则表达式与根据模式接受串的 DFA 之间有很密切的关系，下一节我们将探讨这个关系。但首先需要更仔细地学习 DFA 表示的精确算法，这是因为希望最终能将这些算法变成扫描程序的代码。

我们早已注意到 DFA 的图表并不能表示出 DFA 所需的所有东西而仅仅是给出其运算的要点。实际上，我们发现数学定义意味着 DFA 必须使每个状态和字符都具有一个转换，而且这些导致出错的转换通常是不在 DFA 的图表中。但即使是数学定义也不能描述出 DFA 算法行为的所有方面。例如在出错时，它并不指出错误是什么。在程序将要到达接受状态时或甚至是在转换中匹配字符时，它也不指出该行为。

进行转换时发生的典型动作是：将字符从输入串中移到属于单个记号（记号串值或记号词）累积字符的字符串中。在达到某个接受状态时的典型的动作则是将刚被识别的记号及相关属性返回。遇到出错状态的典型动作则是在输入中备份（回溯）或生成错误记号。

在关于标识符最早的示例中有许多这里将要描述的行为，所以我们再次回到图 2-4 中。由于某些原因，该图中的 DFA 并没有如希望的那样来自扫描程序的动作。首先，出错状态根本就不是一个真正的错误，而是表示标识符将不被识别（如来自于初始状态）或是已看到的一个分隔符，且现在应该接受并生成标识符记号。我们暂时假设（实际这是正确的操作）有其他的转换可表示来自初始状态的非字母转换。接着指出可看到来自状态 in_id 的分隔符，以及应被生成的一个标识符记号，如图 2-5 所示。

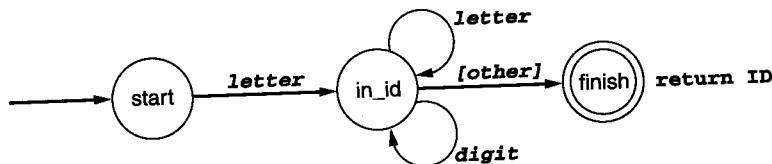
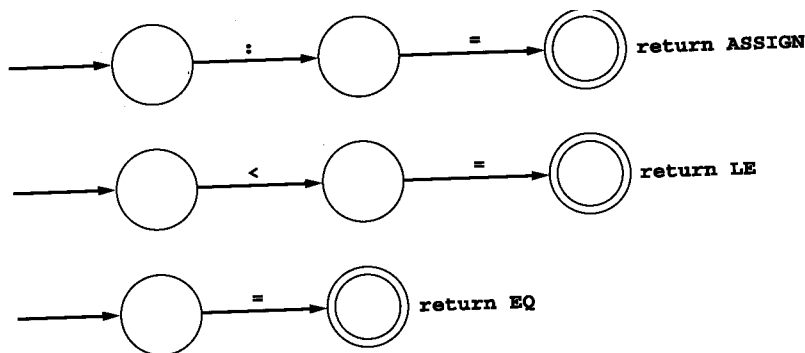


图2-5 有分隔符和返回值的标识符的有穷自动机

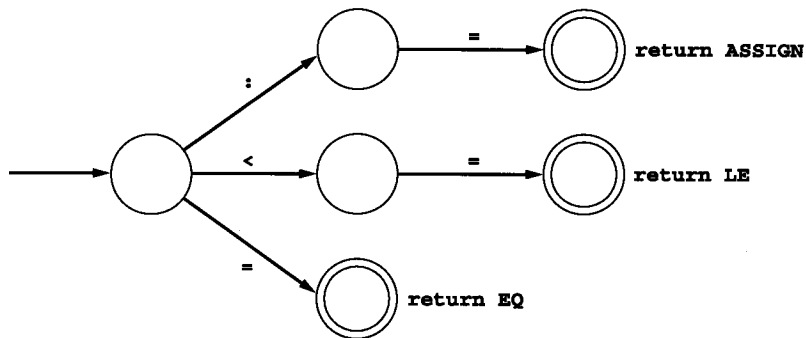
在该图中，*other* 转换前后都带有方括号，它表示了应先行考虑分隔字符，也就是：应先将其返回到输入串并且不能丢掉。此外在该图中，出错状态已变成接受状态，且没有离开接受状态的转换。因为扫描程序应一次识别一个记号并在每一个记号识别之后再一次从它的初始状态开始，所以这正是所需要的。

这个新的图示还表述了在 2.2.4 节中谈到的最长子串原理：DFA 将一直（在状态 `in_id` 中）匹配字母和数字直到找到一个分隔符。与在读取标识符串时允许 DFA 在任何地方接受的旧图相反，我们确实不希望发生某些事情。

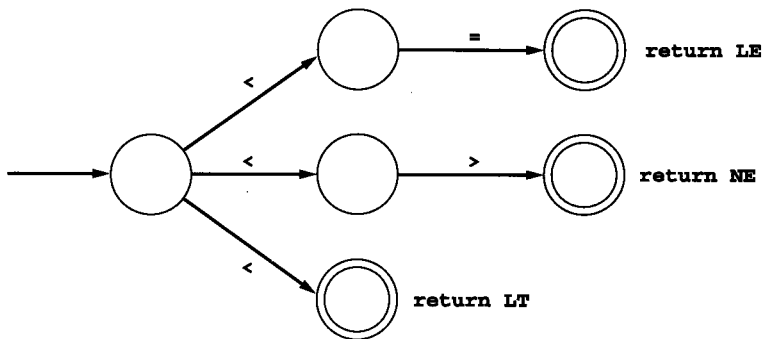
现在将注意力转向如何在一开始就到达初始状态的问题上。典型的程序设计语言中都有许多记号，且每一个记号都能被其自己的 DFA 识别出来。如果这每一个记号都以不同的字符开头，则只需通过将其所有的初始状态统一到一个单独的初始状态上，就能很便利地将它们放在一起了。例如，考虑串：`=`、`<=` 和 `=` 给出的记号。其中每一个都是一个固定串，它们的 DFA 可写作：



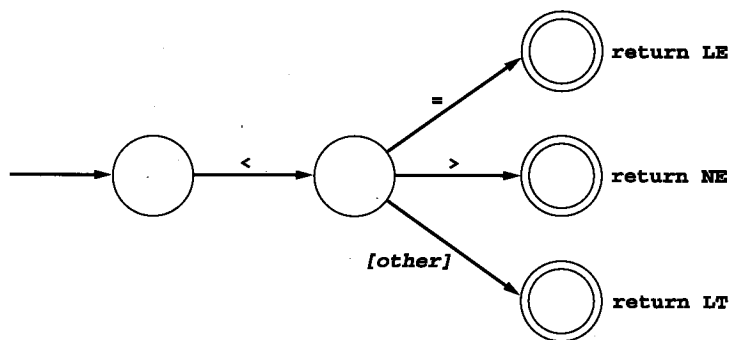
因为每一个记号都是以不同的字符开头的，故只需通过标出它们的初始状态就可得出以下的 DFA：



但是假设有若干个以相同字符开头的记号，例如 `<`、`<=` 和 `<>`，就不能简单地将其表示为如下的图表了。这是因为它不是 DFA（给出一个状态和字符，则通常肯定会有一个指向单个的新状态的唯一转换）：



相反地，我们必须做出安排，以便在每一个状态中都有一个唯一的转换。例如下图：

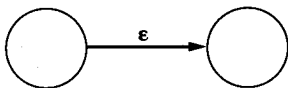


在理论上是应该能够将所有的记号都合并为具有这种风格的一个巨大的 DFA，但是它非常复杂，在使用一种非系统性的方法时尤为如此。

解决这个问题一个方法是将有穷自动机的定义扩展到包括了对某一特定字符一个状态存在有多个转换的情况，并同时为系统地将这些新生成的有穷自动机转换成 NFA 开发一个算法。这里会讲解到这些生成的自动机，但有关转换算法的内容要在下一节才能提到。

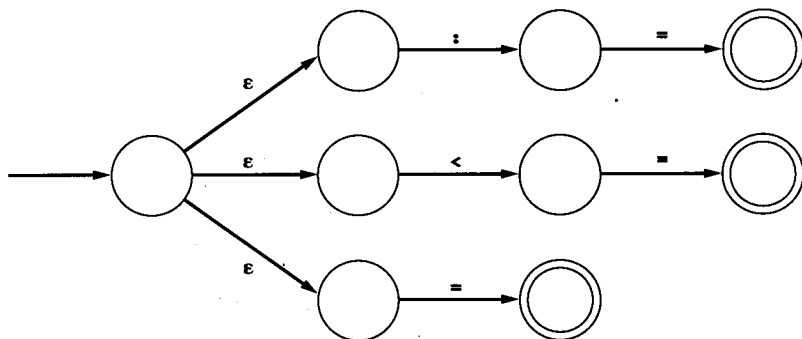
新的有穷自动机称作非确定性有穷自动机 (nondeterministic finite automaton) 或简称为 NFA。在对它下定义之前，还需要为在扫描程序中应用有穷自动机再给出一个概括的讲法： ϵ -转换的概念。

ϵ -转换 (ϵ -transition) 是无需考虑输入串 (且无需消耗任何字符) 就有可能发生的转换。它可看作是一个空串的“匹配”，空串在前面已讲过是写作 ϵ 的。 ϵ -转换在图中的表示就好像 ϵ 是一个真正的字符：



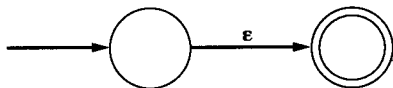
这不应同与在输入中的字符 ϵ 的匹配相混淆：如果字母表包括了这样一个字符，则必须与使用 ϵ 作为表示 ϵ -转换的元字符相区别。

ϵ -转换与直觉有些相矛盾，这是因为它们可以“同时”发生，换言之，就是无需先行和改变到输入串，但它们在两方面有用：首先，它们可以不用合并状态就表述另一个选择。例如：记号 $=$ 、 $<=$ 和 $=$ 的选择可表述为：为每一个记号合并自动机，如下所示：

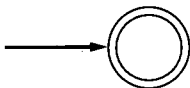


这对于使最初的自动机保持完整并只添加一个新的初始状态来链接它们很有好处。 ϵ -转换

的第2个好处在于它们可以清晰地描述出空串的匹配：

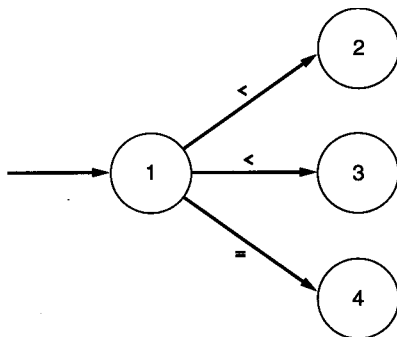


当然，这与下面的DFA等同，该DFA表示接受可在无任何字符匹配时发生：



但是具有前面清晰的表示法也是有用的。

现在来讲述非确定性自动机的定义。它与DFA的定义很相似，但有一点不同：根据上面所讨论的，需要将字母表扩展到包括了 ϵ 。将原来写作的地方（这假设 ϵ 最初并不属于）改写成 $\{\epsilon\}$ （即和 ϵ 的并集）。此外还需要扩展 T （转换函数）的定义，这样每一个字符都可以导致多个状态，通过令 T 的值是状态的一个集合而不是一个单独的状态就可以做到它。例如下表的图表：



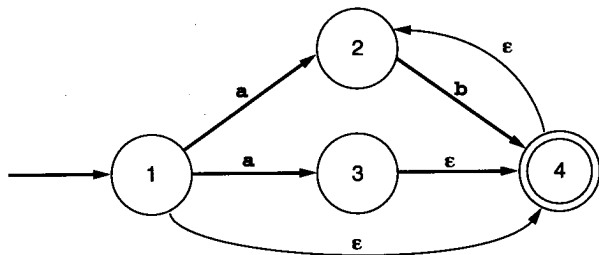
使 $T(1, <) = \{2, 3\}$ ，即：在输入字符 $<$ 上，由状态1可到状态2或状态3，且 T 成为一个将状态/符号对映射到状态集合的函数。因此， T 的范围是状态的 S 集合（ S 的所有子集的集合）的幂集（power set），写作 $\mathcal{P}(S)$ （ S 的草写的 p 的集合）。下面给出定义。

定义：NFA（nondeterministic finite automaton） M 由字母表、状态的集合 S 、转换函数 $T: S \times (\{\epsilon\}) \rightarrow \mathcal{P}(S)$ 、 S 的初始状态 s_0 ，以及 S 的接受状态 A 的集合组成。由 M 接受的语言写作 $L(M)$ ，它被定义为字符 $c_1 c_2 \dots c_n$ ，其中每一个 c_i 都属于 $\{\epsilon\}$ ，且存在关系： s_1 在 $T(s_0, c_1)$ 中、 s_2 在 $T(s_1, c_2)$ 中、 \dots 、 s_n 在 $T(s_{n-1}, c_n)$ 中， s_n 是 A 中的元素。

有关这个定义还需注意以下几点。在 $c_1 c_2 \dots c_n$ 中的任何 c_i 都有可能是 ϵ ，而且真正被接受的串是删除了 ϵ 的串 $c_1 c_2 \dots c_n$ （这是因为 s 和 ϵ 的联合就是 s 本身）。因此，串 $c_1 c_2 \dots c_n$ 中真正的字符数可能会少于 n 个。另外状态序列 $s_1 \dots s_n$ 是从状态集合 $T(s_0, c_1), \dots, T(s_{n-1}, c_n)$ 选出的，这个选择并不总是唯一确定的。实际上这就是为什么称这些自动机是非确定性的原因：接受特定串的转换序列并不由状态和下一个输入字符在每一步中确定下来。实际上，任意个 ϵ 都可在任一点上被引入到串中，并与NFA中 ϵ -转换的数量相对应。因此，NFA并不表示算法，但是却可通过一个在每个非确定性选择中回溯的算法来模拟它，本节下面会谈到这一点。

首先看一些NFA的示例。

例2.10 考虑下面的NFA图。

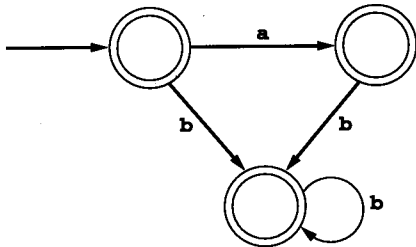


下面两个转换序列都可接受串 abb :

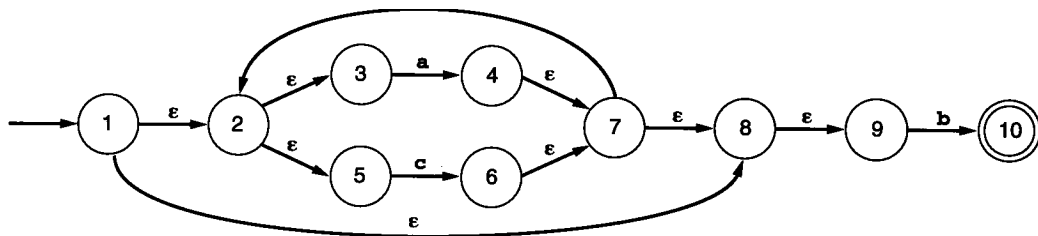
$$\rightarrow 1 \xrightarrow{a} 2 \xrightarrow{b} 4 \xrightarrow{\epsilon} 2 \xrightarrow{b} 4$$

$$\rightarrow 1 \xrightarrow{a} 3 \xrightarrow{\epsilon} 4 \xrightarrow{\epsilon} 2 \xrightarrow{b} 4 \xrightarrow{\epsilon} 2 \xrightarrow{b} 4$$

实际上, a 上的由状态1向状态2的转换与 b 上的由状态2向状态4的转换均允许机器接受串 ab , 接着再使用由状态4向状态2的转换, 所有的串与正则表达式 ab^+ 匹配。类似地, a 上的由状态1向状态3的转换, 和 ϵ 上的由状态3向状态4的转换也允许接受与 ab^* 匹配的所有串。最后, 由状态1向状态4的 ϵ -转换可接受与 b^* 匹配的所有串。因此, 这个 NFA 接受与正则表达式 $ab^+ | ab^* | b^*$ 相同的语言。能够生成相同的语言的更为简单的正则表达式是 $(a | \epsilon)b^*$ 。下面的 DFA 也接受这个语言:



例2.11 考虑下面的NFA :



它通过下面的转换接受串 $acab$:

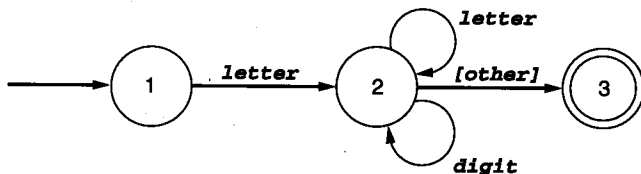
$$\begin{aligned} &\rightarrow 1 \xrightarrow{\epsilon} 2 \xrightarrow{\epsilon} 3 \xrightarrow{a} 4 \xrightarrow{\epsilon} 7 \xrightarrow{\epsilon} 2 \xrightarrow{\epsilon} 5 \xrightarrow{c} 6 \xrightarrow{\epsilon} 7 \\ &\xrightarrow{\epsilon} 2 \xrightarrow{\epsilon} 3 \xrightarrow{a} 4 \xrightarrow{\epsilon} 7 \xrightarrow{\epsilon} 8 \xrightarrow{\epsilon} 9 \xrightarrow{b} 10 \end{aligned}$$

不难看出, 这个 NFA 接受的语言实际上与由正则表达式 $(a | c)^*b$ 生成的语言相同。

2.3.3 用代码实现有穷自动机

将DFA或NFA翻译成代码有若干种方法，本节将会讲到它们。但并不是所有的方法对编译器的扫描程序都有用，本章的最后两节将更详细地讲到与扫描程序相关的编码问题。

请再想一下最开始那个接受由一个字母及一个字母和 / 或数字序列组成的标识符的 DFA 的示例，以及当它位于包含了先行和最长子串原理的修改格式（参见图 2-5）：



模拟这个DFA最早且最简单的方法是在下面的格式中编写代码：

```

{ starting in state 1 }
if the next character is a letter then
  advance the input;
  { now in state 2 }
  while the next character is a letter or a digit do
    advance the input; { stay in state 2 }
  end while;
  { go to state 3 without advancing the input }
  accept;
else
  { error or other cases }
end if;

```

这个代码使用代码中的位置（嵌套于测试中）来隐含状态，这与由注释所指出的一样。如果没有太多的状态（要求有许多嵌套层）且DFA中的循环较小，那么就合适了。类似这样的代码已被用来编写小型扫描程序了。但这个方法有两个缺点：首先它是特殊的，即必须用略微不同的方法处理各个DFA，而且规定一个用这种办法将每个DFA翻译为代码的算法较难。其次：当状态增多或更明确时，且当相异的状态与任意路径增多时，代码会变得非常复杂。现在来考虑一下在例2.9（图2-4）中接受注释的DFA，它可用以下的格式来实现：

```

{ state 1 }
if the next character is "/" then
  advance the input; { state 2 }
if the next character is "*" then
  advance the input ; { state 3 }
done := false;
while not done do
  while the next input character is not "*" do
    advance the input ;
  end while;
  advance the input ; { state 4 }
end if;

```

```

while the next input character is "*" do
  advance the input;
end while;
if the next input character is "/" then
  done := true;
end if;
advance the input;
end while;
accept; { state 5 }
else { other processing }
end if;
else { other processing }
end if;

```

我们注意到这样做复杂性大大增加了，并且还需要利用布尔变量 *done* 来处理涉及到状态 3 和状态 4 的循环。

一个较之好得多的实现方法是：利用一个变量保持当前的状态，并将转换写成一个双层嵌套的 case 语句而不是一个循环。其中第 1 个 case 语句测试当前的状态，嵌套着的第 2 层测试输入字符及所给状态。例如，前一个标识符的 DFA 可翻译为程序清单 2-1 的代码模式。

程序清单 2-1 利用状态变量和嵌套的 case 测试实现标识符 DFA

```

state := 1; { start }
while state = 1 or 2 do
  case state of
    1: case input character of
        letter : advance the input;
           state := 2;
        else state := ... { error or other };
      end case;
    2: case input character of
        letter, digit: advance the input;
           state := 2; { actually unnecessary }
        else state := 3;
      end case;
  end case;
end while;
if state = 3 then accept else error ;

```

请注意这个代码是如何直接反映 DFA 的：转换与对 *state* 变量新赋的状态相对应，并提前输入（除了由状态 2 到状态 3 的“非消耗”转换）。

现在 C 注释的 DFA（图 2-4）可被翻译成程序清单 2-2 中更可读的代码模式。除了这个结构之外，还可使外部 case 基于在输入字符之上，并使内部 case 基于在当前状态之上（参见练习）。

程序清单 2-2 图 2-4 中 DFA 的实现

```

state := 1; { start }
while state = 1, 2, 3 or 4 do

```

```

case state of
1: case input character of
    "/" : advance the input;
        state := 2;
    else state := ... { error or other };
    end case;
2: case input character of
    "*" : advance the input;
        state := 3;
    else state := ... { error or other };
    end case;
3: case input character of
    "*" : advance the input;
        state := 4;
    else advance the input { and stay in state 3 };
    end case;
4: case input character of
    "/" : advance the input;
        state := 5;
    "*" : advance the input; { and stay in state 4 }
    else advance the input;
        state := 3;
    end case;
end case;
end while;
if state = 5 then accept else error ;

```

在前面的示例中，DFA已正好被“硬连”进代码之中，此外还有可能将DFA表示为数据结构并写成实现来自该数据结构的行为的“类”代码。转换表（transition table），或二维数组就是符合这个目标的简单数据结构，它由表示转换函数 T 值的状态和输入字符来索引：

	字母表 C 中的字符
状态 s	代表转换 $T(s, c)$ 的状态

例如：标识符的DFA可表示为如下的转换表：

状态 \ 输入	字母	数字	其他
1	2		
2	2	2	3
3			

在这个表格中，空表项表示未在DFA图中显示的转换（即：它们表示到错误状态或其他过程的转换）。另外还假设列出的第1个状态是初始状态。但是，这个表格尚未指出哪些状态正在接受以及哪些转换不消耗它们的输入。这个信息可被保存在与表示表格相同的数据结构中，或是保存在另外的数据结构中。如果将这些信息添加到上面的转换表中（另用一系列来指出接受状态并用括号指出“未消耗输入”的转换），就会得到下面这个表格：

状态 \ 输入	字母	数字	其他	接受
1	2			不
2	2	2	[3]	不
3				是

又如：下面是C注释的DFA表格（前述的第2个例子）：

状态 \ 输入	/	*	其他	接受
1	2			不
2		3		不
3	3	4	3	不
4	5	4	3	不
5				是

现在若给定了恰当的数据结构和表项，就可以在一个将会实现任何 DFA 的格式中编写代码了。下面的代码图解假设了转换被保存在一个转换数组 *T* 中，而 *T* 由状态和输入字符索引；先行输入的转换（即：那些在表格中未被括号标出的）是由布尔数组 *Advance* 给出，它们也由状态和输入字符索引；而由布尔数组 *Accept* 给出的接受状态则由状态索引。下面就是代码图解：

```

state := 1;
ch := next input character;
while not Accept[state] and not error (state) do
    newstate := T[state, ch];
    if Advance[state, ch] then ch := next input char;
    state := newstate;
end while;
if Accept[state] then accept;

```

类似于刚刚讨论过的算法方法被称作表驱动（table driven），这是因为它们利用表格来引导算法的过程。表驱动方法有若干优点：代码的长度缩短了，相同的代码可以解决许多不同的问题，代码也较易改变（维护）了。但也有一些缺点：表格会变得非常大，使得程序要求使用的空间也变得非常大。实际上，我们刚描述过的数组中的许多空间都是浪费了的。因此，尽管表压缩经常会多耗费时间，但表驱动方法经常仍要依赖于诸如稀疏数组表示法的压缩方法，这是因为扫描程序的效率必须很高，因此尽管可能会在诸如 Lex 的扫描程序生成器程序上用到它们，也是仍很少用到这些方法。在这里也就不再提它们了。

最后注意到可用与 DFA 相似的方法来实现 NFA，但有一点除外——因为 NFA 是非确定性的，所以必须要尝试转换潜在的许多不同序列。因此模拟 NFA 的程序必须储存还未尝试过的转换并回溯失败的转换。除了是由输入串引导搜索之外，这与在指示图中试图找到路径的算法相类似。由于此时进行回溯的算法有可能效率不高，而扫描程序对效率的要求又必须尽可能地高，所以也就不再谈这个算法了。相反地，NFA 的模拟问题可以通过使用下一节将谈到的“将 NFA 转换成 DFA”的方法解决，在这一节中还将还会简要地再谈到 NFA 的模拟问题。

2.4 从正则表达式到DFA

在本节中，我们将学到将正则表达式翻译成 DFA 的算法。由于也存在着将 DFA 翻译成正则表达式的算法，所以这两种概念是等同的。然而因为正则表达式的简洁性，它们趋向于将 DFA 当作记号来描述，而这样扫描程序的生成就通常是从正则表达式开始，并通过 DFA 的构造以达到最终的扫描程序。正是因为这一点，我们只是将兴趣放在完成该等同推导的算法之上。

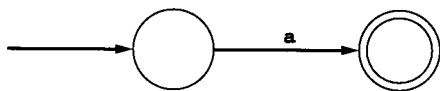
将正则表达式翻译成 DFA 的最简单算法是通过中间构造，在它之中，正则表达式派生出一个 NFA，接着就用该 NFA 构造一个同等的 DFA。现在有一些算法可将正则表达式直译为 DFA，但是它们很复杂，且对中间构造也有些影响。因此我们只关心两个算法：一个是将正则表达式翻译成 NFA，另一个是将 NFA 翻译成 DFA。再与将 DFA 翻译成前一节中描述的程序的算法合并，则构造一个扫描程序的自动过程可分为 3 步，如下所示：



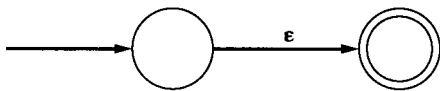
2.4.1 从正则表达式到NFA

下面将要谈到的结构是 Thompson 结构 (Thompson construction)，它以其发明者命名。Thompson 结构利用 ϵ -转换将正则表达式的机器片段“粘在一起”以构成与整个表达式相对应的机器。因此该结构是归纳的，而且它依照了正则表达式定义的结构：为每个基本正则表达式展示一个 NFA，接着再显示如何通过连接子表达式的 NFA（假设这些是已经构造好的）得到每个正则表达式运算。

1) 基本正则表达式 基本正则表达式格式 a 、 ϵ 或 ϕ ，其中 a 表示字母表中单个字符的匹配， ϵ 表示空串的匹配，而 ϕ 则表示根本不是串的匹配。与正则表达式 a 等同的 NFA（即在其语言中准确接受）的是：

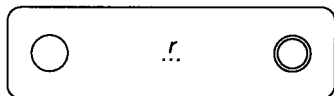


类似地，与 ϵ 等同的 NFA 是：



正则表达式 ϕ 的情况（它在实际的编译器中是不可能发生）将留在练习中。

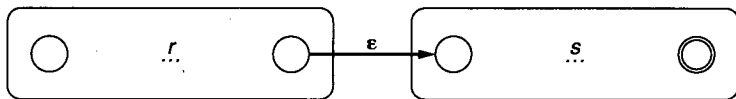
2) 并置 我们希望构造一个与正则表达式 rs 等同的 NFA，其中 r 和 s 都是正则表达式。假设已构造好了与 r 和 s 等同的 NFA，并用 NFA 对应 r 且与 s 类似来写出它：



在该图中，圆角矩形的左边圆圈表示初始状态，右边的双圆表示接受状态，中间的 3 个点表示 NFA 中未显示出的状态和转换。这个图假设与 r 相应的 NFA 中只有一个接受状态。如果构造的每个 NFA 都有一个接受状态，那么这个假设就要调整一下。对于基本正则表达式的 NFA，这是

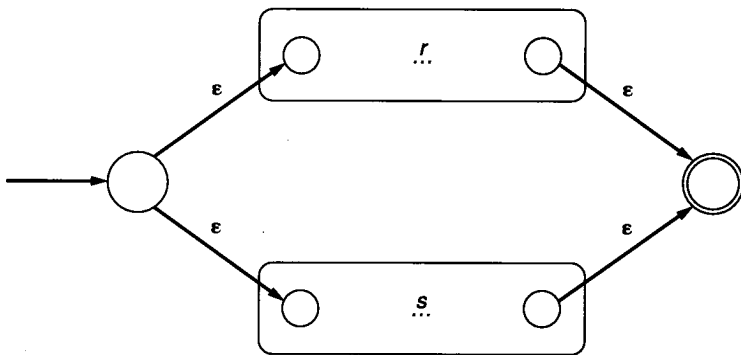
正确的；且对于下面每个结构，它也是正确的。

可将与 rs 对应的NFA构造如下：



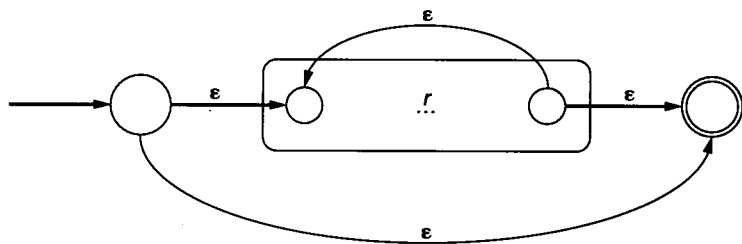
我们已将 r 机的接受状态与 s 机的接受状态通过一个 ϵ -转换连接在一起。新机器将 r 机的初始状态作为它的初始状态，并将 s 机的接受状态作为它的接受状态。很明显，该机可接受 $L(rs) = L(r)L(s)$ 的关系，它也对应于正则表达式 rs 。

3) 在各选项中选择 我们希望在与前面相同的假设下构造一个与 $r|s$ 相对应的NFA。如下进行：



我们添加了一个新的初始状态和一个新的接受状态，并利用 ϵ -转换将它们连接在一起。很明显，该机接受语言 $L(r|s) = L(r) \cup L(s)$ 。

4) 重复 我们需要构造与 r^* 相对应的机器，现假设已有一台与 r 相对应的机器。那么就如下进行：



这里又添加了两个新的状态：一个初始状态和一个接受状态。该机中的重复由从 r 机的接受状态到它的初始状态的新的 ϵ -转换提供。它允许 r 机来回多次移动。为了保证也能接受空串（即 r 的重复为零），就必须也画出一个由新初始状态到新接受状态的 ϵ -转换。

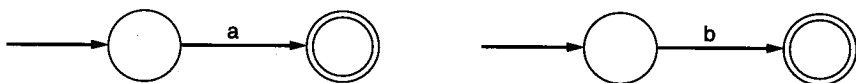
这样就完成了Thompson结构的描述。请读者注意这个构造并不唯一。特别是当将正则表达式运算翻译成NFA时，也可能有另一个结构。例如在表述并置 rs 时，就可以省略在 r 机和 s 机之间的 ϵ -转换，相反却是将 r 机的接受状态等同于 s 机的初始状态，如下：



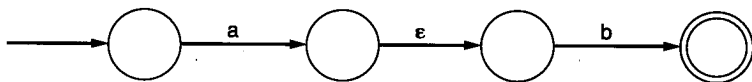
(但是这种简化却要取决于：在别的结构中，接受状态没有来自其他状态的转换，参见练习)。在其他情况下也会有别的简化。之所以像上面那样来表述转换是因为机器构造遵循的原则也非常简单。首先，每个状态具有最多两个来自它的状态，而且如果有两个转换，就必须都是 ϵ -转换。其次，不能在构造时删除状态，而且除了来自接受状态的其他转换之外，转换也不可更改。这些属性就将过程简化了。

用以下几个示例来结束对Thompson结构的讨论。

例2.12 根据Thompson 结构将正则表达式 $ab|a$ 翻译为NFA。首先为正则表达式 a 和 b 分别构造机器：



接着再为并置 ab 构造机器：



现在再为 a 构造另一个机器复件，并利用它们组成 $ab|a$ 完整的NFA，如图2-6所示：

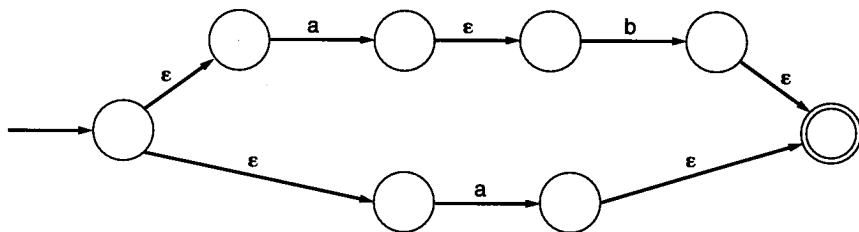
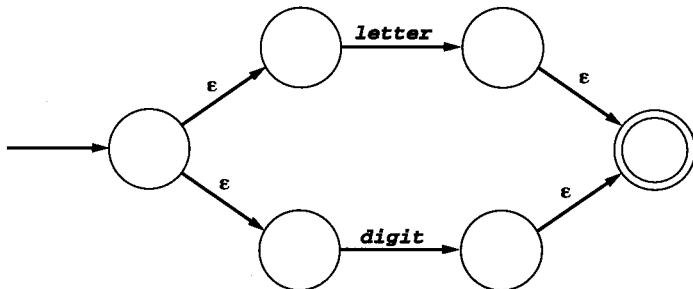


图2-6 利用Thompson结构完成的正则表达式 $ab|a$ 的NFA

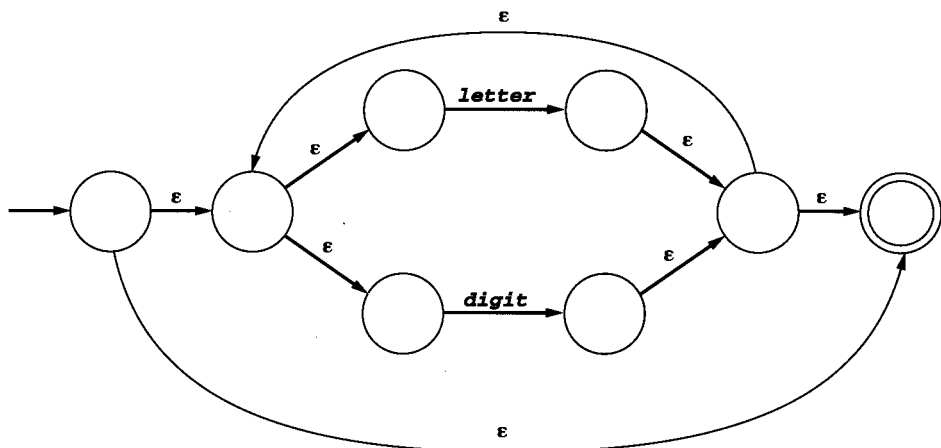
例2.13 利用Thompson 结构完成正则表达式 $letter(letter|digit)^*$ 的NFA。同前例一样，首先分别为正则表达式 $letter$ 和 $digit$ 构建机器：



接着再为选择 $letter|digit$ 构造机器：



现在为重复 $(letter|digit)^*$ 构造NFA，如下所示：



最后，将 `letter` 和 `(letter|digit)*` 并置在一起，并构造该并置的机器以得到完整的 NFA，如图 2-7 所示：

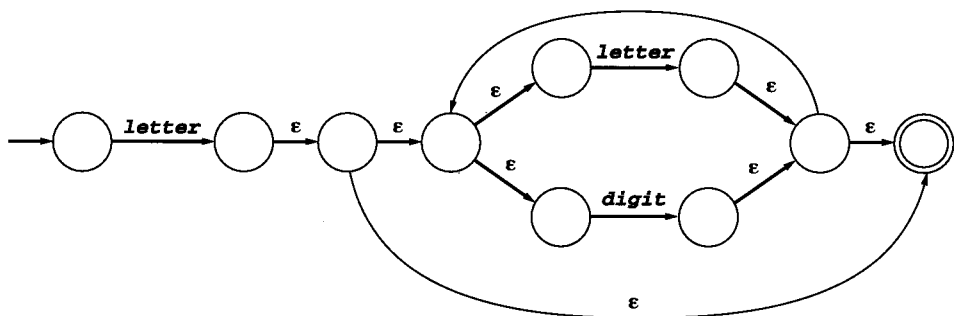


图2-7 利用Thompson结构得到正则表达式 `letter(letter|digit)*` 的NFA

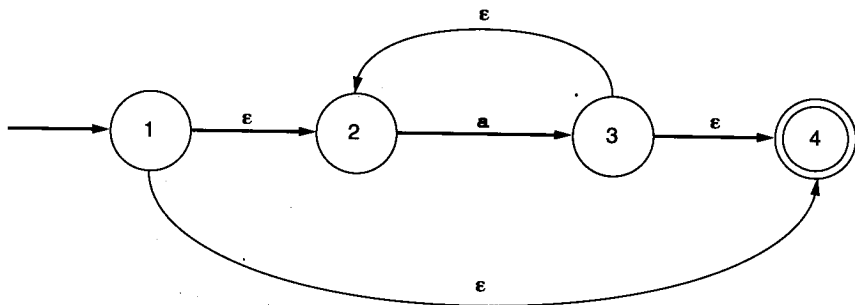
作为最后一个示例：我们注意到例 2.11（2.3.2节）与正则表达式 `(a|c)*b` 在Thompson结构下完全对应。

2.4.2 从NFA到DFA

若给定了一个任意的 NFA，现在来描述构造等价的 DFA（即：可准确接受相同串的 DFA）的算法。为了做到它，则需要可从单个输入字符的某个状态中去除 ϵ - 转换和多重转换的一些方法。消除 ϵ - 转换涉及到了 ϵ - 闭包的构造。 ϵ - 闭包（ ϵ -closure）是可由 ϵ - 转换从某状态或某些状态达到的所有状态集合。消除在单个输入字符上的多重转换涉及跟踪可由匹配单个字符而达到的状态的集合。这两个过程都要求考虑状态的集合而不是单个状态，因此，当看到构建的 DFA 如同它的状态一样，也有原始 NFA 的状态集合时，就不会感到意外了。所以就将这个算法称作子集构造（subset construction）。我们首先较详细地讨论一下 ϵ - 闭包，然后再描述子集构造。

1) 状态集合的 ϵ - 闭包 我们将单个状态 s 的 ϵ - 闭包定义为可由一系列的零个或多个 ϵ - 转换能达到的状态集合，并将这个集合写作 \bar{s} 。该定义的更为数学化的语言将放在练习中，现在直接谈一个示例；但请大家应注意到一个状态的 ϵ - 闭包总是包含着该状态本身。

例2.14 考虑在Thompson结构下，下面与正则表达式 `a*` 相对应的NFA：



在这个NFA，有 $\bar{1} = \{1,2,4\}$, $\bar{2} = \{2\}$, $\bar{3} = \{2,3,4\}$, $\bar{4} = \{4\}$ 。

现在将状态的一个集合的 ϵ -闭包定义为每个单独状态的 ϵ -闭包的和。若 S 是状态集，则用符号表示就是：

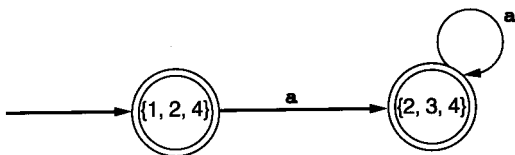
$$\bar{S} = \bigcup_{s \in S} \bar{s}$$

例如在例2.14的NFA中， $\overline{\{1,3\}} = \bar{1} \cup \bar{3} = \{1,2,4\} \cup \{2,3,4\} = \{1,2,3,4\}$ 。

2) 子集构造 现在来描述从一个给定的NFA—— M 来构造DFA的算法，并将其称作 \bar{M} 。首先计算 M 初始状态的 ϵ -闭包，它就变成 \bar{M} 的初始状态。对于这个集合以及随后的每个集合，计算 a 字符之上的转换如下所示：假设有状态的 S 集和字母表中的字符 a ，计算集合 $S'_a = \{t \mid \text{对于 } S \text{ 中的一些 } s, \text{ 在 } a \text{ 上有从 } s \text{ 到 } t \text{ 的转换}\}$ 。接着计算 \bar{S}'_a ，它是 S'_a 的闭包。这就定义了子集构造中的一个新状态和一个新的转换 $S \xrightarrow{a} \bar{S}'_a$ ，继续这个过程直到不再产生新的状态或转换。当接受这些构造的状态时，按照包含了 M 的接受状态的方式作出记号。这就是DFA的 \bar{M} ，它并不包括 ϵ -转换，这是因为每个状态都被构造成了一个 ϵ -闭包。它至多包括了一个来自字母 a 上的状态的转换，这又是因为每个新状态都由从单个字符 a 上的一个状态的转换构造为来自 M 的所有可接受到的状态。

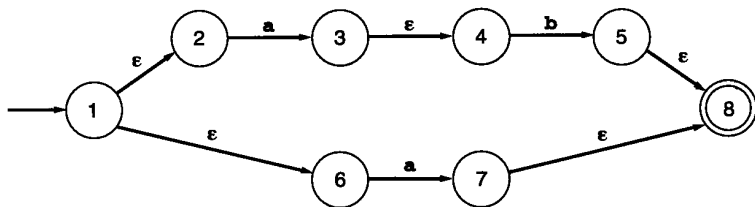
下面用若干示例来说明子集构造。

例2.15 请考虑例2.14中的NFA：与它相对应的DFA的初始状态是 $\bar{1} = \{1,2,4\}$ ，且存在着在字符 a 上的由状态2向状态3的转换，而在 a 上则没有来自状态1或状态4的转换，因此在 a 上就有从 $\{1,2,4\}$ 到 $\overline{\{1,2,4\}}_a = \bar{3} = \{2,3,4\}$ 的转换。由于再也没有来自一个字符上的1、2或4状态的转换了，因此就可将注意力转向新状态 $\{2,3,4\}$ 。此时在 a 上有从状态2到状态3的转换，且也没有来自3或4状态的 a -转换，因此就有从 $\{2,3,4\}$ 到 $\overline{\{2,3,4\}}_a = \bar{3} = \{2,3,4\}$ 的转换，因而也就有从 $\{2,3,4\}$ 到它本身的 a -转换。我们已将所有的状态都考虑完了，所以也构造出了整个DFA。唯一需要读者注意的是NFA的状态4是接受的，这是因为 $\{1,2,4\}$ 和 $\{2,3,4\}$ 都包含了状态4，它们都是相应的DFA的接受状态。将构造出的DFA画出来，其中用状态各自的子集命名状态：

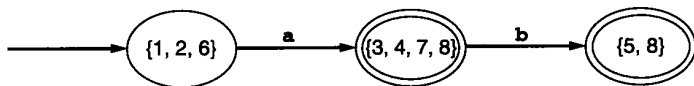


(一旦构造完成，则如果愿意就可将子集术语丢置一旁了)。

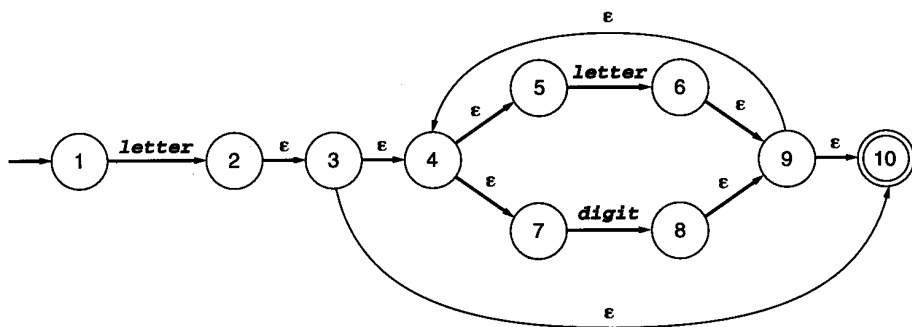
例2.16 考虑向图2-6中的NFA增添状态数：



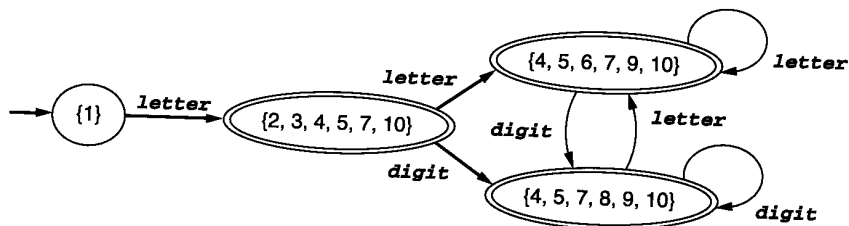
DFA子集结构与它的初始状态 $\overline{1} = \{1, 2, 6\}$ 相同。此时在 a 上有从状态 2 到状态 3 的转换，且有从状态 6 到状态 7 的转换，因此， $\overline{1, 2, 6}_a = \overline{3, 7} = \{3, 4, 7, 8\}$ ，且有 $\{1, 2, 6\} \xrightarrow{a} \{3, 4, 7, 8\}$ 。由于再也没有来自 1、2 或 6 状态的其他字符的转换，则只需看 $\{3, 4, 7, 8\}$ 就行了。此时在 b 上有从状态 4 到状态 5 的转换，且 $\overline{3, 4, 7, 8}_b = \overline{5} = \{5, 8\}$ ，且有转换 $\{3, 4, 7, 8\} \xrightarrow{b} \{5, 8\}$ 。除此之外就再也没有别的转换了。因此所产生的以下 DFA 子集构造与前一个 NFA 等同：



例2.17 考虑图2-7中的NFA（正则表达式 `letter(letter|digit)*` 的Thompson 结构）：



子集构造过程如下：它的初始状态是 $\overline{1} = \{1\}$ ，在 `letter` 上有到 $\overline{2} = \{2, 3, 4, 5, 7, 10\}$ 的转换。在 `letter` 上还有一个从这个状态到 $\overline{6} = \{4, 5, 6, 7, 9, 10\}$ 的转换以及在 `digit` 上有到 $\overline{8} = \{4, 5, 7, 8, 9, 10\}$ 的转换。最后，所有这些状态都有在 `letter` 和 `digit` 上的转换，或是到其自身或是到另一个。完整的DFA在下图中给出：



2.4.3 利用子集构造模拟NFA

上一节简要地讨论了编写模拟NFA的程序的可能性，这是一个要求处理机器的非确定性或非算法本质的问题。模拟NFA的一种方法是使用子集构造，但并非是构造与DFA相关的所有状态，而是在由下一个输入字符指出的每个点上只构造一个状态。因此，这样只构造了在给出的

输入串上被取用的DFA的路径中真正发生的状态集合。这样做的好处在于有可能就不再需要构造整个DFA了，它的缺点在于如果路径中包含了循环，则有可能会多次构造某个状态。

例如：在例2.16中，若使输入串只由单个字符 a 组成，则构建初始状态 $\{1,2,6\}$ 和第2个状态 $\{3,4,7,8\}$ ，之后再移至这个状态并匹配 a 。由于随后再没有 b 了，因此也就无需生成状态 $\{5,8\}$ 了。

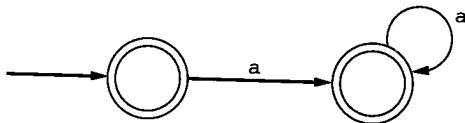
另一方面，在例2.17中，给定了输入串 $r2d2$ ，就有下面的状态和转换序列：

$$\begin{aligned} \{1\} &\xrightarrow{r} \{2, 3, 4, 5, 7, 10\} \xrightarrow{2} \{4, 5, 7, 8, 9, 10\} \\ &\xrightarrow{d} \{4, 5, 6, 7, 9, 10\} \xrightarrow{2} \{4, 5, 7, 8, 9, 10\} \end{aligned}$$

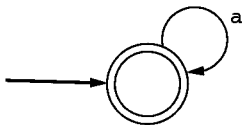
如果在转换发生时构建这些状态，则也构造了 DFA 的所有状态，且构造两次状态 $\{4,5,7,8,9,10\}$ 。因此，这个过程比第一次构造整个 DFA 的效率要低一些。正是由于这个原因，在扫描程序中并不做 NFA 的模拟。但在编辑器和搜索程序中却保留了模式匹配的选项，在编辑器和搜索程序中，正则表达式可由用户动态地提供。

2.4.4 将DFA中的状态数最小化

我们上面所描述的由正则表达式利用算法派生出 DFA 的过程有一个缺点：生成的 DFA 可能比需要的要复杂得多。例如在例2.15中派生出的 DFA：



对于正则表达式 a^* ，下面的DFA同样是可以的：



因为在扫描程序中，效率是很重要的，如果可能的话，在某种意义上构造的 DFA 应最小。实际上，自动机理论中有一个很重要的结果，即：对于任何给定的 DFA，都有一个含有最少量状态的等价的 DFA，而且这个最小状态的 DFA 是唯一的（重命名的状态除外）。人们有可能从任何指定的 DFA 中直接得到这个最小状态的 DFA，本节将简要地描述这个算法，但我们不证明它确实构造了最小状态的等价的 DFA（对于读者而言，通过阅读算法证明一下并不难）。

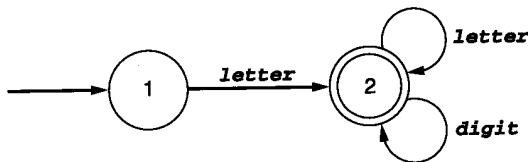
该算法通过创建统一到单个状态的状态集来进行。它以最乐观的假设开始：创建两个集合，其中之一包含了所有的接受状态，而另一个则由所有的非接受状态组成。假设这样来划分原始 DFA 的状态，还要考虑字母表中每个 a 上的转换。如果所有的接受状态在 a 上都有到接受状态的转换，那么这样就定义了一个由新接受状态（所有旧接受状态的集合）到其自身的 a -转换。类似地，如果所有的接受状态在 a 上都有到非接受状态的转换，那么这也定义了由新接受状态到新的非接受状态（所有旧的非接受状态的集合）的 a -转换。另一方面，如果接受状态 s 和 t 在 a 上有转换且位于不同的集合，则这组状态不能定义任何 a -转换，此时就称作 a 区分（distinguish）了状态 s 和 t 。在这种情况下必须根据考虑中状态集合（即所有接受状态的集合）的 a -转换的位置而将它们分隔开。当然状态的每个其他集合都有类似的语句，而且一旦要考虑字母表中的所有字符时，就必须移到它们的位置之上。当然如果还要分隔别的集合，就得返回到开头并重复这一过程。我们继续将原始 DFA 的各部分状态集中到集合里，并一直持续到所有集合只有一个

元素（在这种情况下，就显示原始DFA为最小）或一直是到再没有集合可以分隔了。

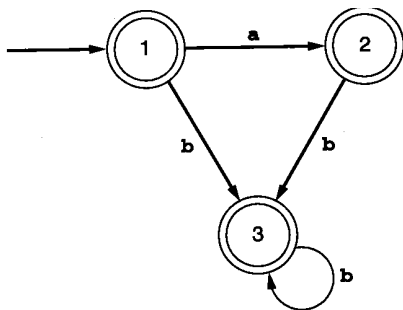
如要准确地完成上面所描述的过程，还必须掌握非接受的错误状态的错误转换。也就是：如果有两个接受状态 s 和 t ，其中 s 有一个到其他接受状态的 a -转换，而 t 却根本没有 a -转换（即：错误转换），那么 a 就将 s 和 t 区分开来了。类似地，如果非接受状态 s 有到某个接受状态的 a -转换，而另一个非接受状态 t 却没有 a -转换，那么在这种情况下， a 也将 s 和 t 区分开来。

下面用几个示例来总结一下状态最小化的讨论。

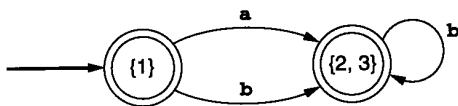
例2.18 考虑在前例中构造的DFA，其与正则表达式 $\text{letter}(\text{letter}|\text{digit})^*$ 相对应，它有4个状态：1个初始状态和3个接受状态，这3个接受状态在 letter 和 digit 上都有到其他接受状态的转换，且除此之外再也没有其他（非错的）转换了。因此，任何字符也不能区分这3个接受状态，且最小化算法将3个接受状态合并为一个接受状态，而剩下了下面的最小状态DFA（即在2.3节开头所看到的）：



例2.19 考虑下面的DFA，在例2.1（2.3.2节）中已指出它与正则表达式 $(a|\varepsilon)b^*$ 相对应：



在这种情况下，所有的状态（除了错误状态之外）都在接受。现在考虑字符 b 。每个接受状态都有到其他接受状态的 b -转换，因此 b 不能区分任何状态。另一方面，状态1存在有到一个接受状态的 a -转换，但状态2和状态3却没有 a -转换（或说成是：在 a 上到错误的非接受状态的错误转换，更合适一些）。所以， a 可将状态1与状态2和3区分开来，我们必须将状态重新分配为集合 $\{1\}$ 和 $\{2,3\}$ 。然后再重复一遍。集合 $\{1\}$ 不能再分隔了，我们也不再考虑它了。状态2和状态3不能由 a 或 b 区分开来。因此，就得到了最小状态的DFA：



2.5 TINY扫描程序的实现

现在开发扫描程序的真正代码以阐明本章前面所学到的概念。在这里将用到 TINY 语言，它曾在第1章（1.7节）中被提到过，接着再讲一些由该扫描程序引出的实际问题。

2.5.1 为样本语言TINY实现一个扫描程序

第1章只是非常简要地介绍了一下TINY语言。现在的任务是完整地指出TINY的词法结构，也就是：定义记号和它们的特性。TINY的记号和记号类都列在表2-1中。

TINY的记号分为3个典型类型：保留字、特殊符号和“其他”记号。保留字一共有8个，它们的含义类似（尽管直到很后面才需知道它们的语义）。特殊符号共有10种：分别是4种基本的整数运算符、2种比较符号（等号和小于），以及括号、分号和赋值符号。除了赋值符号是两个字符的长度之外，其余均为一个字符。

表2-1 TINY语言的记号

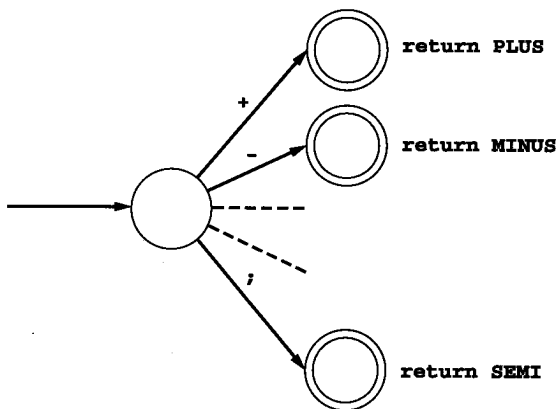
保 留 字	特 殊 符 号	其 他
if	+	数
then	-	(1个或更多的数字)
else	*	
end	/	
repeat	=	
until	<	标识符
read	((1个或更多的字母)
write)	
	;	
	:=	

其他记号就是数了，它们是一个或多个数字以及标识符的序列，而标识符又是（为了简便）一个或多个字母的序列。

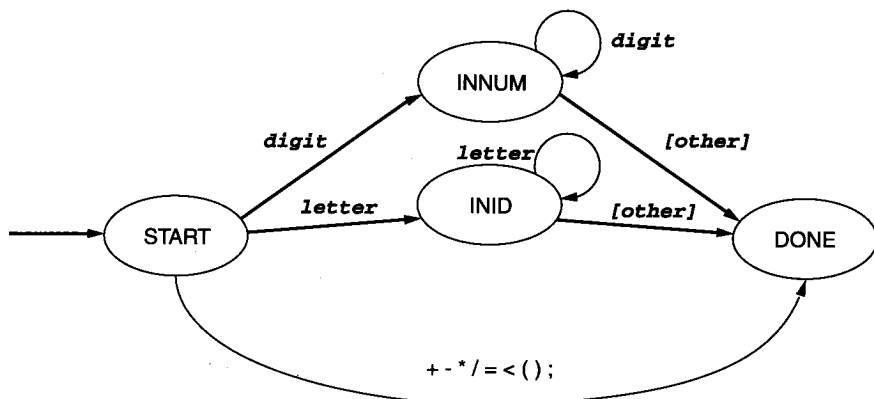
除了记号之外，TINY还要遵循以下的词法惯例：注释应放在花括号{...}中，且不可嵌套；代码应是自由格式；空白格由空格、制表位和新行组成；最长子串原则后须接识别记号。

在为该语言设计扫描程序时，可以从正则表达式开始并根据前一节中的算法来开发NFA和DFA。实际上，前面已经给出了数、标识符和注释的正则表达式（TINY具有更为简单的版本）。其他记号的正则表达式都是固定串，因而均不重要。由于扫描程序的DFA记号十分简单，所以无需按照这个例程就可直接开发这个DFA了。我们将按以下步骤进行。

首先要注意到除了赋值符号之外，其他所有的特殊符号都只有一个字符，这些符号的DFA如下：



在该图中，不同的接受状态是由扫描程序返回的记号区分开来。如果在这个将要返回的记号（代码中的一个变量）中使用其他指示器，则所有接受状态都可集中为一个状态，称之为 **DONE**。若将这个二状态的DFA与接受数和标识符的DFA合并在一起，就可得到下面的DFA：



请注意，利用方括号指出了不可被消耗的先行字符。

现在需要在这个DFA中添加注释、空白格和赋值。一个简单的从初始状态到其本身的循环要消耗空白格。注释要求一个额外的状态，它由花括号左边达到并在花括号右边返回到它。赋值也需要中间状态，它由分号上的初始状态达到。如果后面紧跟有一个等号，那么就会生成一个赋值记号。反之就不消耗下一个字符，且生成一个错误记号。实际上，未列在特殊符号中的所有单个字符既不是空白格或注释，也不是数字或字母，它们应被作为错误而接受，我们将它们与单个字符符号混合在一起。图2-8是为扫描程序给出的最后一个DFA。

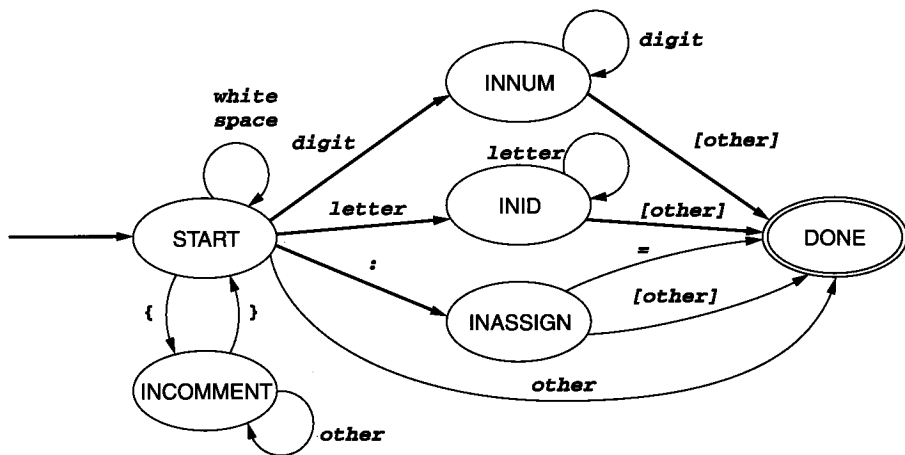


图2-8 TINY扫描程序的DFA

在上面的讨论或图2-8中的DFA都未包括保留字。这是因为根据DFA的观点，而认为保留字与标识符相同，以后在在接受后的保留字表格中寻找标识符是最简单的。当然，最长子串原则保证了扫描程序唯一需要改变的动作是被返回的记号。因而，仅在识别了标识符之后才考虑保留字。

现在再来讨论实现这个DFA的代码，它已被放在了 `scan.h` 文件和 `scan.c` 文件之中（参

见附录B)。其中最主要的过程是`getToken` (第674行到第793行),它消耗输入字符并根据图2-8中的DFA返回下一个被识别的记号。这个实现利用了在第2.3.3节中曾提到过的双重嵌套情况分析,以及一个有关状态的大型情况列表,在大列表中的是基于当前输入字符的单独列表。记号本身被定义成`globals.h` (第174行到第186行)中的枚举类型,它包括在表2-1中列出的所有记号以及内务记号`EOF` (当达到文件的末尾时)和`ERROR` (当遇到错误字符时)。扫描程序的状态也被定义为一个枚举类型,但它是位于扫描程序之中 (第612行到第614行)。

扫描程序还需总地计算出每个记号的特性 (如果有的话),并有时会采取其他动作 (例如将标识符插入到符号表中)。在TINY扫描程序中,所要计算的唯一特性是词法或是被识别的记号的串值,它位于变量`tokenString`之中。这个变量同`getToken`一并提供给编译器其他部分的唯一的两个服务,它们的定义已被收集在头文件`scan.h` (第550行到第571行)。请读者注意声明了`tokenString`的长度固定为41,因此那个标识符也就不能超过40个字符 (加上结尾的空字符)。后面还会提到这个限制。

扫描程序使用了3个全程变量:文件变量`source`和`listing`,在`globals.h`中声明且在`main.c`中被分配和初始化的整型变量`lineno`。

由`getToken`过程完成的额外的簿记如下所述:表`reservedWords` (第649行到第656行)和过程`reservedLookup` (第658行到第666行)完成位于由`getToken`的主要循环识别的标识符之后的保留字的查找,`currentToken`的值也随之改变。标志变量`save`被用作指示是否将一个字符增加到`tokenString`之上;由于需要包括空白格、注释和非消耗的先行,所以这些都是必要的。

到扫描程序的字符输入由`getNextChar`函数 (第627行到第642行)提供,该函数将一个256-字符缓冲区内部的`lineBuf`中的字符取到扫描程序中。如果已经耗尽了这个缓冲区,且假设每一次都获取了一个新的源代码行 (以及增加的`lineno`),那么`getNextChar`就利用标准的C过程`fgets`从`source`文件更新该缓冲区。虽然这个假设允许了更简单的代码,但却不能正确地处理行的字数超过255个字符的TINY程序。在练习中,我们再探讨在这种情况下的`getNextChar`的行为 (以及它更进一步的行为)。

最后,TINY中的数与标识符的识别要求从`INNUM`和`INID`到最终状态的转换都应是非消耗的 (参见图2-8)。可以通过提供一个`ungetNextChar`过程 (第644行到第647行)在输入缓冲区中反填一个字符来完成这一任务,但对于源行很长的程序而言,这也不是很好,练习将提到其他的方法。

作为TINY扫描程序行为的解释,读者可考虑一下程序清单2-3中TINY程序`sample.tny` (在第1章中已作为一个示例给出了)。程序清单2-4假设将这个程序作为输入,那么当`TraceScan`和`EchoSource`都是集合时,它列出了扫描程序的输出。

本节后面将详细讨论由这个扫描程序的实现所引出的一些问题。

程序清单2-3 TINY语言中的样本程序

```
{ Sample program
  in TINY language -
  computes factorial
}
read x; { input an integer }
if 0 < x then { don't compute if x <= 0 }
  fact := 1;
  repeat
    fact := fact * x;
```

```
x := x - 1
until x = 0;
write fact { output factorial of x }
end
```

程序清单2-4 当程序清单2-3中的TINY程序作为输入时，扫描程序的输出

```
TINY COMPILATION: sample.tny
1: { Sample program
2:   in TINY language -
3:   computes factorial
4: }
5: read x; { input an integer }
5: reserved word: read
5: ID, name= x
5: ;
6: if 0 < x then { don't compute if x <= 0 }
6: reserved word: if
6: NUM, val= 0
6: <
6: ID, name= x
6: reserved word: then
7:   fact := 1;
7: ID, name= fact
7: :=
7: NUM, val= 1
7: ;
8:   repeat
8: reserved word: repeat
9:     fact := fact * x;
9: ID, name= fact
9: :=
9: ID, name= fact
9: *
9: ID, name= x
9: ;
10:    x := x - 1
10: ID, name= x
10: :=
10: ID, name= x
10: -
10: NUM, val= 1
11:   until x = 0;
11: reserved word: until
11: ID, name= x
11: =
11: NUM, val= 0
11: ;
12:   write fact { output factorial of x }
12: reserved word: write
12: ID, name= fact
13: end
13: reserved word: end
14: EOF
```

2.5.2 保留字与标识符

TINY对保留字的识别是通过首先将它们看作是标识符，之后再在保留字表中查找它们来

完成的。这在扫描程序中很平常，但它却意味着扫描程序的效率须依赖于在保留字表中查找过程的效率。我们的扫描程序使用了一种非常简便的方法——线性搜索，即按顺序从开头到结尾搜索表格。这对于小型表格不成问题，例如 TINY 中的表格，它只有 8 个保留字，但对于真实语言而言，这却是不可接受的，因为它通常有 30~60 个保留字。这时就需要一个更快的查找，而这又要求使用更好的数据结构而不是线性列表。假若保留字列表是按字母表的顺序写出的，那么就可以使用二分搜索。另一种选择是使用杂凑表，此时我们希望利用一个冲突性很小的杂凑函数。由于保留字不会改变（至少不会很快地），所以可事先开发出这样一个杂凑函数，它们在表格中的位置对于编译器的每一步运行而言都是固定的。人们已经确定了各种语言的最小完善杂凑函数（minimal perfect hash function），也就是说能够区分出保留字且具有最小数值的函数，因此杂凑表可以不大于保留字的数目。例如，如果只有 8 个保留字，则最小完善杂凑函数总会生成一个 0~7 的值，且每个保留字也会生成不同的值（参见“注意与参考”一节）。

在处理保留字时，另一个选择是使用储存标识符的表格，即：符号表。在过程开始之前，将所有的保留字整个输入到该表中并且标上“保留”（因此不允许重新定义）。这样做的好处在于只要求一个查找表。但在 TINY 扫描程序中，直到扫描阶段之后才构造符号表，因此这个方法对于这种类型的设计并不合适。

2.5.3 为标识符分配空间

TINY 扫描程序设计中的另一个缺点是记号串最长仅为 40 个字符。由于大多数的记号的大小都是固定的，所以对于它们而言这并不是问题；但是对于标识符来讲就麻烦了，这是因为程序设计语言经常要求程序中的标识符长度为任意值。更糟的是：如果为每一个标识符都分配一个 40 个字符长度的数组，那么就会浪费掉大多数的空间；这是因为绝大多数的标识符都很短。由于使用了实用程序函数 `copyString` 复制记号串，其中 `copyString` 函数动态地分配仅为所需的（如同将在第 4 章看到的一样），TINY 编译器的代码就不会出现这个问题了。`TokenString` 长度限制的解决办法与之类似：仅仅基于需要来分配，有可能使用 `realloc` 标准 C 函数。另一种办法是为所有的标识符分配最初的大型数组，接着再在该数组中按照自己做的的方式进行存储器的分配（这是将在第 7 章要谈到的标准动态存储器管理计划的特殊情况）。

2.6 利用 Lex 自动生成扫描程序

本节将重复前一节完成的用于 TINY 语言的扫描程序的开发，但此次是利用 Lex 扫描程序生成器从作为正则表达式的 TINY 记号的描述中生成一个扫描程序。由于 Lex 存在着多个不同的版本，所以我们的讨论仅限于对于所有的或大多数版本均通用的特征。Lex 最常见的版本是 `flex`（Fast Lex），它是由 Free Software Foundation 创建的 `Gnu compiler package` 的一部分，可以在许多 Internet 站点上免费得到。

Lex 是一个将包含了正则表达式的文本文件作为其输入的程序，此外还包括每一个表达式被匹配时所采取的动作。Lex 生成一个包含了定义过程 `yylex` 的 C 源代码的输出文件，其中 `yylex` 是与输入文件相对应的 DFA 表驱动的实现，它的运算与 `getToken` 过程类似。接着编译通常称作 `lex.yy.c` 或 `lexyy.c` 的 Lex 输出文件，并将它们链接到一个主程序上以得到一个可运行的程序，这与在前一节中将 `scan.c` 文件与 `tiny.c` 文件链接相似。

下面将首先讨论用于编写正则表达式的 Lex 惯例和 Lex 输入文件的格式，之后还会谈到附录

B中给出的TINY扫描程序的Lex输入文件。

2.6.1 正则表达式的Lex约定

Lex 约定与2.2.3节中所谈到的十分相似，但它并不列出所有的 Lex 元字符且不逐个地描述它们，我们将给出一个概述，之后再在一个表格中写出 Lex 约定。

Lex允许匹配单个字符或字符串，只需像前面各节中所做地一样按顺序写出字符即可。Lex 还允许通过将字符放在引号中而将元字符作为真正的字符来匹配。引号可用于并不是元字符的字符前后，但此时的引号却毫无意义。因此，在要被直接匹配的所有字符前后使用引号很有意义，而不论该字符是否为元字符。例如，可以用 `if` 或 `"if"` 来匹配一个 `if` 语句开始的保留字 `if`。另一方面，如要匹配一个左括号，就必须写作 `"("`，这是因为左括号是一个元字符。另一个方法是利用反斜杠元字符 `\`，但它只有在单个元字符时才起作用：如要匹配字符序列 `(*`，就必须重复使用反斜杠，写作 `\(*`。很明显，`"(*"` 更好一些。另外将反斜杠与正规字符一起使用就有了特殊意义。例如：`\n` 匹配一新行，`\t` 匹配一个制表位（这些都是典型的C约定，大多数这样的约定在Lex中也可行）。

Lex按通常的方法解释元字符 `*`、`+`、`(`、`)` 和 `|`。Lex 还利用问号作为元字符指示可选部分。为了说明前面所讲到的Lex表示法，可为 `a` 串和 `b` 串的集合写出正则表达式，其中这些串是以 `aa` 或 `bb` 开头，末尾则是一个可选的 `c`：

```
(aa|bb)(a|b)*c?
```

或写作：

```
("aa"|"bb")("a"|"b")*"c"?
```

字符类的Lex 约定是将字符类写在方括号之中。例如 `[abxz]` 就表示 `a`、`b`、`x` 或 `z` 中的任意一个字符，此外还可在Lex 中将前面的正则表达式写作：

```
(aa|bb)[ab]*c?
```

在这个格式的使用中还可利用连字符表示出字符的范围。因此表达式 `[0-9]` 表示在Lex 中，任何一个从0~9的数字。句点也是一个表示字符集的元字符：它表示除了新行之外的任意字符。互补集合——也就是不包含某个字符的集合——也可使用这种表示法：将插入符 `^` 作为括号中的第1个字符，因此 `[^0-9abc]` 就表示不是任何数字且不是字母 `a`、`b` 或 `c` 中任何一个的其他任意字符。

例：为一个标有符号的数集写出正则表达式，这个集合可能包含了一个小数部分或一个以字母E开头的指数部分（在2.2.4节中，这个表达式的写法略有不同）：

```
("+"|"-" )?[0-9]+( "."[0-9]+ )?(E("+"|"-" )?[0-9]+ )?
```

Lex有一个古怪的特征：在方括号（表示字符类）中，大多数的元字符都丧失了其特殊状况，且无需用引号引出。甚至如果可以首先将连字符列出来的话，则也可将其写作正则字符。因此，可将前一个数字的正则表达式 `("+"|"-")` 写作 `[-+]`，（但不可写作 `[+-]`，这是因为元字符 `" - "` 用于表示字符的一个范围）。又例如：`[. " ?]` 表示了句号、引号和问号3个字符中的任一个字符（这3个字符在括号中都失去了它们的元字符含义）。但是一些字符即使是在方括号中也仍旧是元字符，因此为了得到真正的字符就必须在字符前加一个反斜杠（由于引号已失去了它们的元字符含义，所以不能用它），因此 `[\\ ^ \]` 就表示了真正的字符 `^` 或 `\`。

Lex中一个更为重要的元字符约定是用花括号指出正则表达式的名字。在前面已经提到过可以为正则表达式起名，而且只要没有递归引用，这些名字也可使用在其他的正则表达式中。

例如：将2.2.4节中的`signedNat`定义如下：

```
nat = [0-9] +  
signedNat = ("+" | "-")? nat
```

在本例和其他示例中，我们使用斜体字将名字和普通的字符序列区分开来。但是 Lex 文件是普通的文本文件，因此无需使用斜体字。相反地，Lex 却遵循将前面定义的名字放在花括号中的约定。因此，上一例在 Lex 中就表示为（Lex 还在定义名字时与等号一起分配）：

```
nat [0-9] +  
signedNat (+|-)? {nat}
```

请注意，在定义名字时并未出现花括号，它只在使用时出现。

表2-2是讨论过的Lex元字符约定的小结列表。Lex 中还有许多我们用不到的元字符，这里也就不讲了（参见本章末尾的“注意与参考”）。

表2-2 Lex中的元字符约定

格 式	含 义
<code>a</code>	字符 <code>a</code>
<code>"a"</code>	即使 <code>a</code> 是一个元字符，它仍是字符 <code>a</code>
<code>\a</code>	当 <code>a</code> 是一个元字符时，为字符 <code>a</code>
<code>a*</code>	<code>a</code> 的零次或多次重复
<code>a+</code>	<code>a</code> 的一次或多次重复
<code>a?</code>	一个可选的 <code>a</code>
<code>a b</code>	<code>a</code> 或 <code>b</code>
<code>(a)</code>	<code>a</code> 本身
<code>[abc]</code>	字符 <code>a</code> 、 <code>b</code> 或 <code>c</code> 中的任一个
<code>[a-d]</code>	字符 <code>a</code> 、 <code>b</code> 、 <code>c</code> 或 <code>d</code> 中的任一个
<code>[^ab]</code>	除了 <code>a</code> 或 <code>b</code> 外的任一个字符
<code>.</code>	除了新行之外的任一个字符
<code>{xxx}</code>	名字 <code>xxx</code> 表示的正则表达式

2.6.2 Lex 输入文件的格式

Lex输入文件由3个部分组成：定义（definition）集、规则（rule）集以及辅助程序（auxiliary routine）集或用户程序（user routine）集。这3个部分由位于新一行第1列的双百分号分开，因此，Lex输入文件的格式如下所示：

```
{definitions}  
%%  
{rules}  
%%  
{auxiliary routines}
```

为了正确理解Lex如何解释这样的输入文件，就必须记住该文件的一些部分是正则表达式信息，Lex利用这个信息指导构成它的C输出代码，而文件的另一部分则是提供给Lex的真正的C代码，Lex会在适当的位置逐字地将它插入到输出代码中。在我们逐个解释完这3个部分以及给出一些示例之后，将会告诉大家Lex在这里所用的规则。

定义部分出现在第1个双百分号之前。它包括两样东西：第1件是必须插入到应在这一部分

中分隔符“%{”和“%}”之间的任何函数外部的任意C代码（请注意这些字符的顺序）。第2件是正则表达式的名字也得在该部分定义。这个名字的定义写在另一行的第1列，且其后（后面有一个或多个空格）是它所表示的正则表达式。

第2个部分包含着一些规则。它们由一连串带有C代码的正则表达式组成；当匹配相对应的正则表达式时，这些C代码就会被执行。

第3个部分包括着一些C代码，它们用于在第2个部分被调用且不在任何地方被定义的辅助程序。如果要将Lex输出作为独立程序来编译，则这一部分还会有一个主程序。当第2个双百分号无需写出时，就不会出现这一部分（但总是需要写出第1个百分号）。

下面给出一些示例来说明Lex输入文件的格式。

例2.20 以下的Lex输入指出一个给文本添加行号的扫描程序，它将其输出发送到屏幕上（如果被重定向，则是一个文件）：

```
%{
/* a Lex program that adds line numbers
   to lines of text, printing the new text
   to the standard output
*/
#include <stdio.h>
int lineno = 1;
}%
line *.\n
%%
{line} { printf ( "%5d %s", lineno++, yytext ); }
%%
main()
{ yylex(); return 0; }
```

例如，运行从这个输入文件本身的Lex中获取的程序会得到以下输出：

```
1 %{
2 /* a Lex program that adds line numbers
3    to lines of text, printing the new text
4    to the standard output
5 */
6 #include <stdio.h>
7 int lineno = 1;
8 %}
9 line *.\n
10 %%
11 { line } { printf ( "%5d %s", lineno++,yytext); }
12 %%
13 main ( )
14 { yylex( ) ; return 0; }
```

下面为这个使用了这些行号的Lex 输入文件作出解释。首先，第1行到第8行都是位于分隔符%{和%}之间，这样就使这些行可以直接插入到由Lex 产生的C代码中，而它是位于任何过程的外部。特别是从第2行到第5行的注释可以插入到程序开头的附近，还将从外部插入#include指示与第6行和第7行上的整型变量lineno的定义，因此lineno就变成了一个全程变量且在最初被赋值为1。出现在第1个%%之前的其他定义是名字line的定义，line被定义

为正则表达式 `".*\n"`，它与零个或多个其后接有一新行的字符匹配（但不包括新行）。换言之：由 `line` 定义的正则表达式与输入的每一行都匹配。在第 10 行的 `%%` 之后，第 11 行包括了 Lex 输入文件的行为部分。此时每当匹配了一个 `line` 时，都写下了一个要完成的行为（根据 Lex 约定，`line` 前后都用花括号以示与其作为一个名字相区别）。正则表达式之后是 `action`，即每当匹配正则表达式都要被执行的 C 代码。在这个示例中，该行为由包含在一个 C 块的花括号中的 C 语句组成（请记住在名字 `line` 前后的花括号与构成下面行为中的 C 代码块的花括号有完全不同的作用）。这个 C 语句将打印行号（在一个有 5 个空格的范围内且右对齐）以及在它后面要增加 `lineno` 的串 `yytext`。`yytext` 的名字是 Lex 赋予并由正则表达式匹配的串的内部名字，此时的正则表达式是由输入的每一行组成（包括新行）^①。最后，当 Lex 生成 C 代码结束时在第 2 个双百分号（第 13 行和第 14 行）之后插入 C 代码。在本例中，代码包括了一个调用函数 `yylex` 的 `main` 过程（`yylex` 是由 Lex 构造的过程的名字，这个 Lex 实现了与正则表达式和在输入文件的行为部分中与给出的行为相关的 DFA）。

例 2.21 考虑下面的 Lex 输入文件：

```
%{
/* a Lex program that changes all numbers
   from decimal to hexadecimal notation,
   printing a summary statistic to stderr
*/
#include <stdlib.h>
#include <stdio.h>
int count = 0;
}%
digit [0-9]
number {digit}+
%%
{number} { int n = atoi (yytext);
          printf ("%x", n);
          if (n > 9) count++;}

%
main( )
{ yylex ( );
  fprintf ( stderr, "number of replacements = %d",
           count);

  return 0 ;
}
```

它在结构上与前例类似，但 `main` 过程打印了在调用了 `yylex` 之后替换到 `stderr` 的次数。这个例子与前例的不同还在于它并没有匹配所有的文本。而实际上只在行为部分匹配了数字；在行为部分中，行为的 C 代码第 1 次将匹配串（`yytext`）转变成一个整型 `n`，接着又将其打印为十六进制的格式（`printf ("%x", ...)`），最后如果这个数大于 9 则增加 `count`（如果小于或等于 9，则与在十六进制中没有分别）。因此，为串指定的唯一行为就是数字序列。Lex 还生成了一个可匹配所有非数字字符的程序，并将它传送到输出中。这是 Lex 的一个缺省行为（default action）的示例。如果字符或字符串与行为部分中的任何一个正则表达式都不匹配，则缺省地，

① 本节最后将用一个表格列出这一节中所讨论过的 Lex 内置名字。

Lex 将会匹配它并将它返回到输出中 (Lex 还可被迫生成一个程序错误, 但这里不讨论它了)。Lex 的内部定义宏 **ECHO** 也可特别指定缺省行为 (下一个示例将会学到它)。

例2.22 考虑下面的Lex 输入文件：

```
%{
/* Selects only lines that end or
   begin with the letter 'a'.
   Deletes everything else.
*/
#include <stdio.h>
}%
ends_with_a .*a\n
begins_with_a a.*\n
%%
{ends_with_a} ECHO;
{begins_with_a} ECHO;
.*\n ;
%%
main( )
{ yylex( ); return 0; }
```

这个Lex 输入将以字符 *a* 开头或结尾的所有输入行均写到输出上, 并消除其他行。行的消除是由 **ECHO** 规则下的规则引起的, 在这个规则中, 为 C 行为代码编写一个分号就可为正则表达式 *.*\n* 指定“空”行为。

这个Lex 输入还有一个值得注意的特征：所列的规则具有二义性 (ambiguous), 这是因为串可匹配多个规则。实际上, 无论它是否是以 *a* 开头或结尾的行的一部分, 任何输入行都可与表达式 *.*\n* 匹配。Lex 有一个解决这种二义性的优先权系统。首先, Lex 总是匹配可能的最长子串 (因此 Lex 总是生成符合最长子串原则的扫描程序)。接着, 如果最长子串仍与两个或更多的规则匹配, Lex 就选取在行为部分所列的第 1 个规则。正是由于这个原因, 上面的 Lex 输入文件就将 **ECHO** 行为放在第 1 个。如果已按下面的顺序列出行为：

```
.*\n;
{ends_with_a} ECHO;
{begins_with_a} ECHO;
```

则由 Lex 生成的程序就不会再生成任何文件的输出, 这是因为第 1 个规则已匹配了输入的每一行了。

例2.23 在本例中, Lex 生成了将所有的大写字母转变成小写字母的程序, 但这不包括 C- 风格注释中的字母 (即：任何位于分隔符 */*...*/* 之间的字母)：

```
%{
/* Lex program to convert uppercase to
   lowercase except inside comments
*/
#include <stdio.h>
#ifdef FALSE
#define FALSE 0
#endif
#ifdef TRUE
```

```

#define TRUE 1
#endif
%}
%%
[A-Z] {putchar(tolower(yytext[0]));
      /* yytext[0] is the single
        uppercase char found */
      }
"/*" { char c ;
      int done = FALSE;
      ECHO;
      do
      { while ((c=input())!='*')
        putchar(c);
        putchar(c);
        while((c=input())=='*')
        putchar(c);
        putchar(c);
        if (c == '/') done = TRUE;
      } while (!done);
      }

%%
void main(void )
{ yylex();}

```

这个示例显示如何编写代码以回避较难的正则表达式，并且像执行一个 Lex 行为一样直接执行一个小的 DFA。读者可以回忆一下 2.2.4 节中用 C 注释的一个正则表达式极难编写，相反地，我们只为开始 C 注释的串编写了正则表达式即：`"/*"`，之后还提供了搜索结束串 `"*/"` 的行为代码，同时为注释中的其他字符提供适当的行为（此时仅是返回它们而不是继续进行）。这是通过模拟例 2.9 中的 DFA 来完成的（参见图 2-4）。一旦识别出串 `"/*"`，则就是在状态 3，因此代码就在这里找到了 DFA。首先做的事情是在字符中循环直到看到一个星号（与状态 3 中的 *other* 循环相对应）为止，如下所示：

```
while ((c=input())!='*') putchar(c);
```

这里又使用了 Lex 另一个内部过程 `input`，该过程的使用——并不是利用 `getchar` 的一个直接输入——保证使用了 Lex 输入缓冲区，而且还保留了这个输入串的内部结构（但请注意，我们确实使用了一个直接输出过程 `putchar`，2.6.4 节将谈到它）。

DFA 代码的下一步与状态 4 相对应。再次循环直到看不到星号为止；之后如在字符前有一个前斜杠就退出；否则就返回到状态 3 中。

本节的最后是小结在各例中介绍到的 Lex 约定。

(1) 二义性的解决

Lex 输出总是首先将可能的最长子串与规则相匹配。如果某个子串可与两个或更多的规则匹配，则 Lex 的输出将找出列在行为部分中的第 1 个规则。如果没有规则可与任何非空子串相匹配，则缺省行为将下一个字符复制到输出中并继续下去。

(2) C 代码的插入

1) 任何写在定义部分 `%{` 和 `%}` 之间的文本将被直接复制到外置于任意过程的输出程序之中。

2) 辅助过程中的任何文本都将被直接复制到 Lex 代码末尾的输出程序中。3) 将任何跟在行为部

分（在第1个%%之后）的正则表达式之后（中间至少有一个空格）的代码插入到识别过程 `yylex` 的恰当位置，并在与对应的正则表达式匹配时执行它。代表一个行为的 C 代码可以既是一个 C 语句，也可以是一个由任何说明及由位于花括号中的语句组成的复杂的 C 语句。

(3) 内部名字

表2-3列出了在本章中所提到过的 Lex 内部名字，大多数都已在前面的示例中讲过了。

表2-3 一些 Lex 内部名字

Lex 内部名字	含义/使用
<code>lex.yy.c</code> 或 <code>lexyy.c</code>	Lex 输出文件名
<code>yylex</code>	Lex 扫描例程
<code>yytext</code>	当前行为匹配的串
<code>yyin</code>	Lex 输入文件（缺省： <code>stdin</code> ）
<code>yyout</code>	Lex 输出文件（缺省： <code>stdout</code> ）
<code>input</code>	Lex 缓冲的输入例程
<code>ECHO</code>	Lex 缺省行为（将 <code>yytext</code> 打印到 <code>yyout</code> ）

上表中有一个前面未曾提到过的特征：Lex 为一些文件备有其自身的内部名字：`yyin`和 `yyout`，Lex 从这些文件中获得输入并向它们发送输出。通过标准的 Lex 输入例程 `input` 就可自动地从文件 `yyin` 中得到输入。但是在前述的示例中，却回避了内部输出文件 `yyout`，而只通过 `printf` 和 `putchar` 写到标准输出中。一个允许将输出赋到任一文件中的更好的实现方法是用 `fprintf(yyout, ...)` 和 `putc(..., yyout)` 取代它们。

2.6.3 使用 Lex 的 TINY 扫描程序

附录B中有一个 Lex 输入文件 `tiny.1` 的列表，`tiny.1` 将生成 TINY 语言的扫描程序（2.5 节已描述了 TINY 语言的记号，参见表 2-1）。下面对这个输入文件（第 3000 行到第 3072 行）做一些说明。

首先，在定义部分中，直接插入到 Lex 输出中的 C 代码是由 3 个 `#include` 指示（`globals.h`、`util.h` 和 `scan.h`）及 `tokenString` 特性组成的。在扫描程序和其他的 TINY 编译器之间有必要提供一个界面。

定义部分更深的内容还包括了定义 TINY 记号的正则表达式的名字的定义。请注意，`number` 的定义利用了前面定义的名字 `digit`，而 `identifier` 的定义利用了前面定义的 `letter`。由于新行会导致增加 `lineno`，所以定义还区分了新行和其他的空白格（空格和制表位，以及第 3019 行和第 3020 行）。

Lex 输入的行为部分由各种记号的列表和 `return` 语句组成，其中 `return` 语句返回在 `globals.h` 中定义的恰当记号。在这个 Lex 定义中，在标识符规则之前列出了保留字规则。假若首先列出标识符规则，Lex 的二义性解决规则就会总将保留字识别为标识符。我们还可以写出与前一节中的扫描程序中相同的代码，在这里只能识别出标识符，然后再在表中查找保留字。由于单独识别的保留字使得由 Lex 生成的扫描程序代码中的表格变得很大（而且扫描程序使用的存储器也会因此变得很大），因此在真正的编译中倾向于使用它。

Lex 输入还有一个“怪僻”：即使 TINY 注释的正则表达式很容易书写，也必须编写识别注释的代码以确保正确地更新了 `lineno`。正则表达式实际是：

```
"{ "[^\\}] * " }
```

(请注意, 方括号中的花括号的作用是删除右花括号的元字符含义——引号在这里不起作用)^①。

我们还注意到: 并未为在遇到输入文件末尾时返回 EOF 编写代码。Lex 过程 `yylex` 在遇到 EOF 时有一个缺省行为——它返回 0 值。正是由于这个原因, 在 `globals.h` 中的 `TokenType` 定义 (第 179 行) 中首先写出记号 `ENDFILE`, 所以它有 0 值。

最后, `tiny.1` 文件包括了辅助过程部分中的 `getToken` 过程的定义 (第 3056 行到第 3072 行)。虽然这个代码包含了 Lex 内部代码 (如 `yyin` 和 `yyout`) 的一些在主程序中能更好地直接完成的特殊初始化, 它还是允许直接使用 Lex 生成的扫描程序, 而无需改变 TINY 编译器中的任何其他文件。实际上在生成 C 扫描程序 `lex.yy.c` (或 `lexyy.c`) 后, 就可编译这个文件, 并可将它与其他的 TINY 源文件链接以生成一个基于 Lex 版本的编译器了。但是这个版本的编译器却缺少了以前版本的一项服务, 这是因为没有源代码回应提供的行号 (参见练习 2.35)。

练习

2.1 为以下的字符集编写正则表达式; 若没有正则表达式, 则说明原因:

- 以 *a* 开头和结尾的所有小写字母串。
- 以 *a* 开头或/和结尾的所有小写字母串。
- 第 1 个不为 0 的所有数字串。
- 所有表示偶数的数字串。
- 每个 2 均在每个 9 之前的所有数字串。
- 所有的 *a* 串和 *b* 串, 且不包含 3 个连续的 *b*。
- 包含单数个 *a* 或/和单数个 *b* 的所有 *a* 串和 *b* 串。
- 包含偶数个 *a* 或偶数个 *b* 的所有 *a* 串和 *b* 串。
- a* 和 *b* 数目相等的所有 *a* 串和 *b* 串。

2.2 为由以下正则表达式生成的语言写出英语描述:

- $(a|b)^*a(a|b|\varepsilon)$
- $(A|B|\dots|Z)(a|b|\dots|z)^*$
- $(aa|b)^*(a|bb)^*$
- $(0|1|\dots|9|A|B|C|D|E|F)^+(x|X)$

2.3 a. 许多系统都有 `grep` (global regular expression print) 的一个版本, 它是最早为 Unix 编写的正则表达式搜索程序^②。寻找一个描述你的本地 `grep` 文档, 并描述它的元符号约定。

b. 如果你的编辑器为它的串搜索接受某种正则表达式, 则描述它的元符号约定。

2.4 在正则表达式的定义中, 我们讲了一些运算的优先问题, 但并未提到它们之间的联系。例如并未指出 $a|b|c$ 表示的是 $(a|b)|c$ 还是 $a|(b|c)$ 以及与并置是否相似, 为什么是这样的呢?

2.5 试证明对于任何正则表达式 r 都有 $L(r^{**}) = L(r^*)$ 。

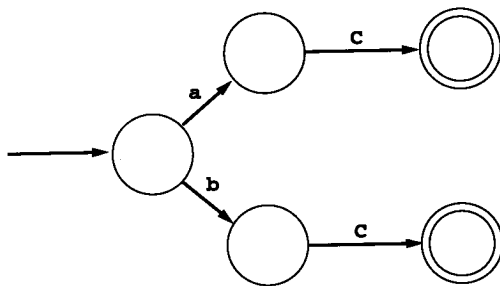
2.6 在使用正则表达式描述的程序设计语言的记号中, 并不需要有元符号 ϕ (空集) 或 ε (空串)。为什么?

① Lex 的一些版本有一个内部定义的变量 `yylineno`, 它可以自动更新。用这个变量代替 `Lineno` 就有可能省掉特殊代码了。

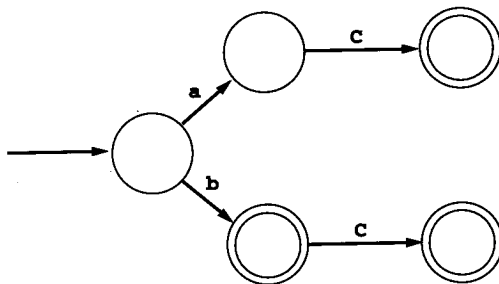
② 大多数的 Unix 系统上实际有 3 个版本的 `grep`: “ regular ” `grep`、`egrep` (extended `grep`) 和 `fgrep` (fast `grep`)。

- 2.7 画出与正则表达式 ϕ 相对应的DFA。
- 2.8 为练习2.1中a至i的每个字符集画出DFA，或说出为什么不存在DFA？
- 2.9 画出从C语言中接受以下4个保留字case、char、const和continue的DFA。
- 2.10 利用将输入字符作为外部情况测试及将状态作为内部情况测试，重写用于C注释的DFA实现（2.3.3节）的伪代码。将你的伪代码与书中的作一比较。什么时候为代码实现DFA使用这个组织？
- 2.11 给NFA的状态集的闭包下一个数学定义。
- 2.12 a. 使用Thompson结构将正则表达式 $(a|b)^*a(a|b|\epsilon)$ 转化成一个NFA。
b. 利用子集结构将a中的NFA转化成一个DFA。
- 2.13 a. 使用Thompson结构将正则表达式 $(aa|b)^*a(a|bb)^*$ 转化成一个NFA。
b. 利用子集结构将a中的NFA转化成一个DFA。
- 2.14 利用子集结构将例2.10（2.3.2节）中的NFA转化成一个DFA。
- 2.15 2.4.1节讲到了为了并置将Thompson结构简化，即：省略被并置的正则表达式的两个NFA之间的 ϵ -转换。另外还提到这种简化在结构的其他步骤中，除接受状态之外不能再有转换。请给出一个例子来说明这一点（提示：考虑用于重复的一个新NFA结构，它省略了新初始状态和接受状态，再为 r^*s^* 考虑NFA）。
- 2.16 为下面的DFA提供2.4.4节中用到的状态最小化算法：

a.



b.



- 2.17 Pascal注释中允许有两个不同的注释约定：花括号对 $\{ \dots \}$ （当在TINY中）和括号-星号对 $(\dots)^*$ 。试写出一个识别这两种风格的注释的DFA。
- 2.18 a. 为Lex表示法中的C注释写出一个正则表达式（提示：参见2.2.3节中的讨论）。
b. 试证明a中的答案是正确的。
- 2.19 下面的正则表达式已作为C注释的一个Lex定义给出（参见Schreiner和Friedman [1985.p.25]）：

$$\frac{1}{n} \sum_{i=1}^n \left(\frac{\lambda_i}{\mu_i} \right) \left| \frac{\lambda_i}{\mu_i} - \frac{\bar{\lambda}}{\bar{\mu}} \right| = \frac{1}{n} \sum_{i=1}^n \left(\frac{\lambda_i}{\mu_i} \right)^2 - \left(\frac{\bar{\lambda}}{\bar{\mu}} \right)^2$$

请说明这个表达式是不正确的（提示：考虑串 `/**_/**/`）。

编程练习

- 2.20 编写将一个C程序中的所有注释字母均大写的程序。
- 2.21 编写一个程序，使之将一个C程序注释之外的所有保留字全部大写（在 Kernighan和 Ritchie [1988,p.192]中可找到一个C的保留字列表）。
- 2.22 编写一个Lex输入文件，使之可生成将C程序中所有注释的字母均大写的程序。
- 2.23 编写一个Lex输入文件，使之可生成将C程序注释之外的所有保留字均大写的程序。
- 2.24 编写一个Lex输入文件，使之生成可计算文本文件的字符、单词和行数且能报告该数字的程序。试定义一个单词是不带标点或空格的字母和 /或数字的序列。标点和空格不计算为单词。
- 2.25 通过能将注释中的行为与其他任何地方的行为区别开来的一个全程标志 `inComment`，可缩短例2.23（2.6.2节中）的Lex代码。按上所述重写这个示例中的代码。
- 2.26 向例2.23中的Lex 代码添加嵌套的C注释。
- 2.27
 - a. 重写TINY的扫描程序，使之能利用二分法搜索保留字。
 - b. 重写TINY的扫描程序，使之能利用杂凑表查找保留字。
- 2.28 通过为 `tokenString` 动态地分配空间来去除对于 TINY扫描程序中的标识符不可多于40个字符的限制。
- 2.29
 - a. 当源程序行超过扫描程序中的缓冲区大小时，测试 TINY扫描程序的行为，同时找出尽可能多的问题。
 - b. 重写TINY扫描程序以解决在a中发现的问题（或至少改善其行为）（此时要求重写 `getNextChar`和`ungetNextChar`过程）。
- 2.30 如果要求在TINY扫描程序中不用 `ungetNextChar`过程来完成非消费的转换，则也可使用布尔标志指出要消费的当前字符，这样就无需在输入中进行备份了。请按照这个方法重写TINY扫描程序，并将它与现存的代码相比较。
- 2.31 利用一个称为 `nestLevel`的计数器将嵌套注释添加到TINY扫描程序中。
- 2.32 在TINY扫描程序中添加 Ada风格的注释（Ada注释以两个连字符开头并一直到行结尾）。
- 2.33 向TINY的Lex扫描程序添加表格中的保留字的备份（可以像在手写的 TINY扫描程序一样使用线性搜索，或使用练习2.27中建议的搜索方法）。
- 2.34 在TINY扫描程序的lex代码中添加 Ada风格注释（Ada注释以两个连字符开头并一直到行结尾）。
- 2.35 在TINY扫描程序的Lex代码中添加源代码行回应（利用 `EchoSource`标志），这样当设置了标志时，源代码的每一行及其行数都会打印到列表文件中（此时要求比本书所提到的更多的Lex内部知识）。

注意与参考

Hopcroft和Ullman [1979]详细讨论了正则表达式的数学理论与有穷自动机，在其中还能找到一些对该理论的历史发展的描述。特别地，这里还有一个关于对有穷自动机与正则表达式相

等价的证明（本章只谈到了这个等式的一个方面）。在这里还可找到关于 pumping 引理的讨论，以及对在描述格式中正则表达式的限制的推理。此外还能看到对状态最小化算法的更详细的描述，另外还包括了对这样的 DFA 在本质上是唯一的证明。在 Aho、Hopcroft 和 Ullman [1986] 中可看到从正则表达式到 DFA 的进一步构造（与这里所谈到的用两步来构造相反）的描述。此外这里还有将表格压缩成一个表驱动的扫描程序的方法。Sedgewick [1990] 中有使用与本章所提到的相当不同的 NFA 约定的 Thompson 结构的描述，此外在这里还能看到为了识别保留字，描述了二分搜索和杂凑法的算法（第 6 章还要讨论到杂凑）。Cichelli [1980] 和 Sager [1985] 都提到了 2.5.2 节中的最小完善杂凑函数。一个称作 `gperf` 的实用程序可作为 Gnu 编译包的一部分来分发，它可以快速地为保留字更大的集合生成完善的杂凑函数。虽然它们并不是最小生成的，但在实际中仍很有用。Schmidt [1990] 中有 `gperf` 的描述。

Lesk [1975] 中有 Lex 扫描程序生成器的最早描述，它仍然对许多最近的版本有点作用。稍后的版本，尤其是 Flex (Paxson [1990]) 解决了一些很重大的问题，它可以与调整得已很好的手写的扫描程序相竞争 (Jacobson [1987])。在 Schreiner 和 Friedman [1985] 中可看到 Lex 的一个有用的描述，以及可完成各种格式匹配任务的简单 Lex 程序的许多示例。在 Kernighan and Pike [1984] 中可找到格式匹配的 `grep` 家族 (练习 2.3) 的简要描述，其更为深入的讨论可在 Aho [1979] 中找到。Landin [1966] 和 Hutton [1992] 讲到了在 2.2.3 节中所提到过的。

第3章 上下文无关文法及分析

本章要点

- 分析过程
- 上下文无关文法
- 分析树与抽象语法树
- 二义性
- 扩展的表示法：EBNF和语法图
- 上下文无关语言的形式特性
- TINY 语言的语法

分析的任务是确定程序的语法，或称作结构，也正是这个原因，它又被称作语法分析（syntax analysis）。程序设计语言的语法通常是由上下文无关（context-free grammar）的文法规则（grammar rule）给出，其方式同扫描程序识别的由正则表达式提供的记号的词法结构相类似。上下文无关文法的确利用了与正则表达式中极为类似的命名惯例和运算。二者的主要区别在于上下文无关文法的规则是递归的（recursive）。例如一般来说，if 语句的结构应允许可其中嵌套其他的if语句，而在正则表达式中却不能这样做。这个区别造成的影响很大。由上下文无关文法识别的结构类比由正则表达式识别的结构类大大增多了。用作识别这些结构的算法也与扫描算法差别很大，这是因为它们必须使用递归调用或显式管理的分析栈。用作表示语言语义结构的数据结构现在也必须是递归的，而不再是线性的（如同用于词法和记号中的一样）了。经常使用的基本结构是一类树，称作分析树（parse tree）或语法树（syntax tree）。

同第2章相似，在学习分析算法和如何利用这些算法进行真正的分析之前需要先学习上下文无关文法的理论，但是又与在扫描程序中的情形不同——其中主要只有一种算法方法（表示为有穷自动机），分析涉及到要在许多属性和能力截然不同的方法中做出选择。按照它们构造分析树或语法树的方式，算法大致可分为两种：自顶向下分析（top-down parsing）和由底向上分析（bottom-up parsing）。对这些分析方法的详细讨论将放到以后的章节中，本章只给出分析过程的一般性描述，之后还要学习上下文无关文法的基础理论。最后一节则从一个上下文无关文法的角度给出TINY语言的文法。对上下文无关文法理论和语法树熟悉的读者可以跳过本章中间的某些内容（或将其作为复习）。

3.1 分析过程

分析程序的任务是从由扫描程序产生的记号中确定程序的语法结构，以及或隐式或显式地构造出表示该结构的分析树或语法树。因此，可将分析程序看作一个函数，该函数把由扫描程序生成的记号序列作为输入，并生成语法树作为它的输出：

记号序列 $\xrightarrow{\text{分析程序}}$ 语法树

记号序列通常不是显式输入参数，但是当分析过程需要下一个记号时，分析程序就调用诸如getToken的扫描程序过程以从输入中获得它。因此，编译器的分析步骤可减为对分析程序的一个调用，如下所示：

```
syntaxTree = parse();
```


在单遍编译中，分析程序合并编译器中所有的其他阶段，这还包括了代码生成，因此也就不需要构造显式的语法树了（分析步骤本身隐式地表示了语法树），由此就发生了一个

```
parse();
```

调用。在编译器中更多的是多遍，此时后面的遍将语法树作为它们的输入。

语法树的结构在很大程度上依赖于语言特定的语法结构。这种树通常被定义为动态数据结构，该结构中的每个节点都由一个记录组成，而这个记录的域包括了编译后面过程所需的特性（即：并不是那些由分析程序计算的特性）。节点结构通常是节省空间的各种记录。特性域还可以是在需要时动态分配的结构，它就像一个更进一步节省空间的工具。

在分析程序中有一个比在扫描程序中更为复杂的问题，这就是对于错误的处理。在扫描程序中，如果遇到一个字符是不正规记号的一部分，那么它只需生成一个出错记号并消耗掉这个讨厌的字符即可（在某种意义上，通过生成一个出错记号，扫描程序就克服了发生在分析程序上的困难）。但对于分析程序而言，它必须不仅报告一个出错信息，而且还须从错误状态恢复（recover）并继续进行分析（去找到尽可能多的错误）。分析程序有时会执行错误修复（error repair），此时它从提交给它的非正确的版本中推断出一个可能正确的代码版本（这通常是在简单情况下才发生的）。错误恢复的一个尤为重要的方面是有意义的错误信息报告以及在尽可能接近真正错误时继续分析下去。由于要到错误真正地已经发生了分析程序才会发现它，所以做到这一点并不简单。由于错误恢复技术依赖于所使用的特定分析算法，所以本章先就不学习它了。

3.2 上下文无关文法

上下文无关文法说明程序设计语言的语法结构。除了上下文无关文法涉及到了递归规则之外，这样的说明与使用正则表达式的词法结构的说明十分类似。例如在一个运算中，就可以使用带有加法、减法和乘法的简单整型算术表达式。这些表达式可由下面的文法给出：

$$\begin{aligned} \text{exp} & \text{ exp op exp } | (\text{exp}) | \text{number} \\ \text{op} & \text{ + } | \text{- } | * \end{aligned}$$

3.2.1 与正则表达式比较

在第2章中为 *number* 给出的正则表达式规则如下所示：

```
number = digit digit*
digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

试与上面上下文无关文法的样本作一比较。基本正则表达式规则有 3 种运算：选择（由竖线元字符表示）、并置（不带元符号）以及重复（由星号元符号提供）。此外还可使用等号来表示正则表达式的名字定义，此时名字用斜体书写以示与真正字符序列的区别。

文法规则使用相似的表示法。名字用斜体表示（但它是一种不同的字体，所以可与正则表达式相区分）。竖线仍表示作为选择的元符号。并置也用作一种标准运算。但是这里没有重复的元符号（如正则表达式中的星号*），稍后还会再讲到它。表示法中的另一个差别是现在用箭头符号“ \rightarrow ”代替了等号来表示名字的定义。这是由于现在的名字不能简单地由其定义取代，而需要更为复杂的定义过程来表示，这是由定义的递归本质决定的^①。在我们的示例中，exp 的

① 参见本章后面的语法规则和等式。

规则是递归的，其中名字 *exp* 出现在箭头的右边。

读者还要注意到文法规则将正则表达式作为部件。在 *exp* 规则和 *op* 规则中，实际有6个表示语言中记号的正则表达式。其中5个是单字符记号：*+*、*-*、***、*(* 和 *)*，另一个是名字 *number*。记号的名字表示数字序列。

与这个例子的格式相类似的文法规则最初是用在 Algol60 语言的描述中。其表示法是由 John Backus 为 Algol60 报告开发，之后又由 Peter Naur 更新，因此这个格式中的文法通常被称作 **Backus-Naur 范式** (Backus-Naur form) 或 **BNF 文法**。

3.2.2 上下文无关文法规则的说明

同正则表达式类似，文法规则是定义在一个字母表或符号集之上。在正则表达式中，这些符号通常就是字符，而在文法规则中，符号通常是表示字符串的记号。在上一章中，我们利用 C 中的枚举类型定义了扫描程序中的记号；本章为了避免涉及到特定实现语言（例如 C）中表示记号的细节，就使用了正则表达式本身来表示记号。此时的记号就是一个固定的符号，如同在保留字 *while* 中或诸如 *+* 或 *:=* 这样的特殊符号一样，使用在第 2 章曾用到的代码字体书写串本身。对于作为表示多于一个串的标识符和数的记号来说，代码字体为斜体，这就同假设这个记号是正则表达式的名字（这是它经常的表示）一样。例如，将 TINY 语言的记号字母表表示为集合

```
{if, then, else, end, repeat, until, read, write,
 identifier, number, +, -, *, /, =, <, (, ), ;, := }
```

而不是记号集（如在 TINY 扫描程序中定义的一样）：

```
{IF, THEN, ELSE, END, REPEAT, UNTIL, READ, WRITE, ID, NUM,
 PLUS, MINUS, TIMES, OVER, EQ, LT, LPAREN, RPAREN, SEMI, ASSIGN }
```

假设有一个字母表，BNF 中的上下文无关文法规则 (context-free grammar rule in BNF) 是由符号串组成。第 1 个符号是结构名字，第 2 个符号是元符号，这个符号之后是一个符号串，该串中的每个符号都是字母表中的一个符号（即一个结构的名字）或是元符号 |。

在非正式术语中，对 BNF 的文法规则解释如下：规则定义了箭头左边名字的结构。这个结构被定义为由被竖线分隔开的选择右边的一个选项组成。每个选项中的符号序列和结构名字定义了结构的布局。例如，前例中的文法规则：

$$\begin{aligned} \text{exp} & \quad \text{exp op exp} \mid (\text{exp}) \mid \text{number} \\ \text{op} & \quad + \mid - \mid * \end{aligned}$$

第 1 个规则定义了一个表达式结构（用名字 *exp*）由带有一个算符和另一个表达式的表达式，或一个位于括号之中的表达式，或一个数组成。第 2 个规则定义一个算符（利用名字 *op*）由符号 *+*、*-* 或 *** 构成。

这里所用的元符号和惯例与广泛使用中的类似，但读者应注意到这些惯例并没有统一的标准。实际上，用以代替箭头元符号的通常有 *=*（等号）、*:*（冒号）和 *::=*（双冒号等号）。在普通文本文件中，找到能替代斜体用法的方法也很有必要，通常的办法是在结构名字前后加尖括号 *<...>* 并将原来斜体的记号名字大写；因此，使用不同的惯例，上面的文法规则就可变为：

```
<exp> ::= <exp> <op> <exp> \ ( <exp> ) \ NUMBER
<op> ::= + \ - \ *
```

每一个作者都还有这些表示法的其他变形。本节后面将会谈到一些非常重要的变形（其中

一些有时也会碰到)。这里要先讲一下有关表示法方面的另外两个较小的问题。

在BNF的元符号中使用括号有时很有用,这同括号可在正则表达式中重新安排优先权很相似。例如,可将上面的文法规则重写为如下一个单一的文法规则:

$$\text{exp} \quad \text{exp} ("+" | "-" | "*") \text{exp} | "(" \text{exp} ")" | \text{number}$$

在这个规则中,括号很必要,它用于将箭头右边的表达式之间的算符选择组合在一起,这是因为并置优先于选择(同在正则表达式中一样)。因此,下面的规则就具有了不同(但不正确)的含义:

$$\text{exp} \quad \text{exp} "+" | "-" | "*" \text{exp} | "(" \text{exp} ")" | \text{number}$$

请读者再留意一下:当将括号包含为一个元符号时,就有必要区分括号记号与元符号,这一点是通过将括号记号放在引号中做到的,这同在正则表达式中的一样(为了具有连贯性,也将算符符号放在引号中)。

由于经常可将括号中的部分分隔成新的文法规则,所以在BNF中的括号并不像元符号一样缺之不可。实际上如果允许在箭头左边的相同名字可出现任意次,那么由竖线元符号给出的选择运算在文法规则中也不是一定要有的。例如,简单的表达式文法可写作:

$$\begin{aligned} \text{exp} & \quad \text{exp op exp} \\ \text{exp} & \quad (\text{exp}) \\ \text{exp} & \quad \text{number} \\ \text{op} & \quad + \\ \text{op} & \quad - \\ \text{op} & \quad * \end{aligned}$$

然而,通常把文法规则写成每个结构的所有选择都可在一个规则中列出来,而且每个结构名字在箭头左边只出现一次。

有时我们需要为说明简便性而给出一些用简短表示法写出的文法规则的示例。在这些情形中,应将结构名字大写,并小写单个的记号符号(它们经常仅仅是一个字符);因此,按这种速记方法可将简单的表达式文法写作:

$$\begin{aligned} E & \quad E O E | (E) | n \\ O & \quad + | - | * \end{aligned}$$

有时当正在将字符如同记号一样使用时,且无需使用代码字体来书写它们,则也可简化表示法如下:

$$\begin{aligned} E & \quad E O E | (E) | a \\ O & \quad + | - | * \end{aligned}$$

3.2.3 推导及由文法定义的语言

现在讨论文法规则如何确定一种“语言”或者是记号的正规串集。

上下文无关文法规则确定了为由规则定义的结构记号符号符合语法的串集。例如,算术表达式

$$(34-3)*42$$

与7个记号的正规串相对应

$$(\text{number} - \text{number}) * \text{number}$$

其中 number 记号具有由扫描程序确定的结构且串本身也是一个正规的表达式,这是因为

每一个部分都与由文法规则

$$\begin{aligned} \text{exp} & \quad \text{exp op exp} \mid (\text{exp}) \mid \text{number} \\ \text{op} & \quad + \mid - \mid * \end{aligned}$$

给出的选择对应。另一方面，串

(34-3*42

就不是正规的表达式，这是因为左括号没有一个右括号与之匹配，且 exp 的文法规则中的第2个选择要求括号需成对生成。

文法规则通过推导确定记号符号的正规串。推导 (derivation) 是在文法规则的右边进行选择的一个结构名字替换序列。推导以一个结构名字开始并以记号符号串结束。在推导的每一个步骤中，使用来自文法规则的选择每一次生成一个替换。

例如，图3-1利用同在上面简单表达式文法中的一样给出的文法规则为表达式 (34-3)*42 提供了一个推导。在每一步中，为替换所用的文法规则选择都放在了右边（还为便于在后面引用为每一步都编了号）。

(1)	$\text{exp} \Rightarrow \text{exp op exp}$	$[\text{exp} \rightarrow \text{exp op exp}]$
(2)	$\Rightarrow \text{exp op number}$	$[\text{exp} \rightarrow \text{number}]$
(3)	$\Rightarrow \text{exp} * \text{number}$	$[\text{op} \rightarrow *]$
(4)	$\Rightarrow (\text{exp}) * \text{number}$	$[\text{exp} \rightarrow (\text{exp})]$
(5)	$\Rightarrow (\text{exp op exp}) * \text{number}$	$[\text{exp} \rightarrow \text{exp op exp}]$
(6)	$\Rightarrow (\text{exp op number}) * \text{number}$	$[\text{exp} \rightarrow \text{number}]$
(7)	$\Rightarrow (\text{exp} - \text{number}) * \text{number}$	$[\text{op} \rightarrow -]$
(8)	$\Rightarrow (\text{number} - \text{number}) * \text{number}$	$[\text{exp} \rightarrow \text{number}]$

图3-1 算术表达式 (34-3)*42 的推导

请注意，推导步骤使用了与文法规则中的箭头元符号不同的箭头，这是由于推导步骤与文法规则有一点差别：文法规则定义 (define)，而推导步骤却通过替换来构造 (construct)。在图3-1的第1步中，串 exp op exp 从规则 exp exp op exp 的（在BNF中对于exp的第1个选择）右边替换了单个的exp。在第2步中，串 exp op exp 中的exp被符号 number 从选择 exp number 的右边替换掉以得到串 exp op number。在第3步中，符号 * 从规则 op * 中替换 op（在BNF中对于op的第3个选择）以得到串 exp * number，等等。

由推导从exp符号中得到的所有记号符号的串集是被表达式的文法定义的语言 (language defined by the grammar)。这个语言包括了所有合乎语法的表达式。可将它用符号表示为：

$$L(G) = \{s \mid \text{exp} \quad *s\}$$

其中G代表表达式文法，s 代表记号符号的任意数组串（有时称为句子 (sentence)），而符号 * 表示由如前所述的替换序列组成的推导（星号用作指示步骤的序列，这与在正则表达式中指示重复很相像）。由于它们通过推导“产生”L(G)中的串，文法规则因此有时也称作产生式 (production)。

文法中的每一个结构名定义了符合语法的记号串的自身语言。例如，在简单表达式文法中由op 定义的语言定义了语言 {+, -, *} 只由3个符号组成。我们通常对由文法中最普通的结构定义的语言最感兴趣。用于程序设计语言的文法经常定义一个称作程序的结构，而该结构的语言是程序设计语言的所有符合语法的程序的集合（注意这里在两个不同的意思中所使用的“语言”）。

例如：Pascal的BNF以诸如

```

program    program-heading ; program-block.
program-heading    ...
program-block    ...
...

```

的文法规则开始（第1个规则认为程序由一个程序头、随后的分号、随后的程序块，以及位于最后的一个句号组成）。在诸如C的带有独立编译的语言中，最普通的结构通常称作编译单元。除非特别指出不是，在各种情况下都假定在文法规则中，第1个列出的就是这个最普通的结构（在上下文无关文法的数学理论中，这个结构称作开始符号（start symbol））。

通过更深一些的术语可更清楚地区分结构名和字母表中的符号（因为它们经常是编译应用程序的记号，所以我们一直调用记号符号）。由于在推导中必须被进一步替换（它们不终结推导），所以结构名也称作非终结符（nonterminal）。相反地，由于字母表中的符号终结推导，所以它们被称作终结符（terminal）。因为终结符通常是编译应用程序中的记号，所以这两个名字在使用时是基本同义的。终结符和非终结符经常都被认作是符号。

下面是一些由文法生成的语言示例。

例3.1 考虑带有单个文法规则的文法 G

$$E \rightarrow (E) \mid a$$

这个文法有1个非终结符 E 、3个终结符 $(,)$ 以及 a 。这个文法生成语言 $L(G) = \{ a, (a), ((a)), (((a))), \dots \} = \{ ({}^n a) \mid n \text{ 是一个 } 0 \text{ 的整型} \}$ ，即：串由零个或多个左括号、后接一个 a ，以及后面是与左括号相同数量的右括号组成。作为这些串的一个推导示例，我们给出 $((a))$ 的一个推导：

$$E \rightarrow (E) \rightarrow ((E)) \rightarrow (((a)))$$

例3.2 考虑带有单个文法规则

$$E \rightarrow (E)$$

的文法 G 。除了减少了选项 $E \rightarrow a$ 之外，这是与前例相同的文法。这个文法根本就不生成串，因此它的语言是空的： $L(G) = \{ \}$ 。其原因在于任何以 E 开头的推导都生成总是含有 E 的串，所以没有办法推导出一个仅包含有终结符的串。实际上，与带有所有递归进程（如归纳论证或递归函数）相同，递归地定义一个结构的文法规则必须总是有至少一个非递归情况（称之为基础情况（base case））。本例中的文法并没有这样的情况，且任何潜在的推导都注定为无穷递归。

例3.3 考虑带有单个文法规则

$$E \rightarrow E + a \mid a$$

的文法 G 。这个文法生成所有由若干个“+”分隔开的 a 组成的串：

$$L(G) = \{ a, a + a, a + a + a, a + a + a + a, \dots \}$$

为了（非正式地）查看它，可考虑规则 $E \rightarrow E + a$ 的效果：它引起串 $+a$ 在推导右边不断地重复：

$$E \rightarrow E + a \rightarrow E + a + a \rightarrow E + a + a + a \rightarrow \dots$$

最后，必须用基础情况 $E \rightarrow a$ 来替换左边的 E 。

若要更正式一些，则可如下来归纳证明：首先，通过归纳 a 的数目来表示每个串 $a + a + \dots + a$ 都在 $L(G)$ 中。推导 $E \rightarrow a$ 表示 a 在 $L(G)$ 中；现在假设 $s = a + a + \dots + a$ 在 $L(G)$ 中，且有 $n-1$ 个 a ，则存在推导 $E \rightarrow s$ 。现在推导 $E \rightarrow E + a \rightarrow s + a$ 表示串 $s + a$ 在 $L(G)$ 中，且其中有 n 个 a 。相反地，我们也表示出在 $L(G)$ 中的任何串 s 都必须属于格式 $a + a + \dots + a$ 。这是通过归纳推导

的长度得出的。假设推导的长度为1, 则它属于格式 $E \rightarrow a$, 而且 s 是正确格式。现在假设以下两个推导都为真: 所有串都带有长度为 $n-1$ 的推导, 并使 $E \rightarrow *s$ 是长度为 $n>1$ 的推导; 则这个推导开头必须将 E 改为 $E+a$, 格式 $E \rightarrow E+a \rightarrow *s' + a = s$ 也是这样。则 s' 有长度为 $n-1$ 的推导, 格式 $a+a+\dots+a$ 也是这样; 因此, s 本身必须也具有这个相同的格式。

例3.4 考虑下面语句的极为简化的文法:

```
statement  if-stmt | other
if-stmt   if (exp) statement
          | if (exp) statement else statement
exp       0 | 1
```

这个文法的语言包括位于类似 C 的格式中的嵌套 if 语句 (我们已将逻辑测试表达式简化为 0 或 1, 且除 if 语句外的所有语句都被放在终结符 **other** 中了)。例如, 这个语言中的串是:

```
other
if ( 0 ) other
if ( 1 ) other
if ( 0 ) other else other
if ( 1 ) other else other
if ( 0 ) if ( 0 ) other
if ( 0 ) if ( 1 ) other else other
if ( 1 ) other else if ( 0 ) other else other
...
```

请注意, if 语句中的可选 else 部分由用于 if-stmt 的文法规则中的单个选择指出。

在前面我们已注意到: BNF 中的文法规则规定了并置和选择, 但不具有与正则表达式的 $*$ 相等价的特定重复运算。由于可由递归得到重复, 所以这样的运算实际并不必要 (如同在功能语言中的程序员所知道的)。例如, 文法规则

$$A \rightarrow Aa \mid a$$

或文法规则

$$A \rightarrow aA \mid a$$

都生成语言 $\{a^n \mid n \text{ 是 } \geq 1 \text{ 的整型}\}$ (具有 1 个或多个 a 的所有串的集合), 该语言与由正则表达式 a^+ 生成的语言相同。例如, 串 $aaaa$ 可由带有推导

$$A \rightarrow Aa \rightarrow Aaa \rightarrow Aaaa \rightarrow aaaa$$

的第 1 个文法规则生成。一个类似的推导在第 2 个文法规则中起作用。由于非终结符 A 作为定义 A 的规则左边的第 1 个符号出现, 所以这些文法中的第 1 个是左递归 (left recursive)[⊖], 第 2 个文法则是右递归 (right recursive)。

例3.3是左递归文法规则的另一个示例, 它引出串 “ $+a$ ” 的重复。可将本例及前一个示例归纳如下。考虑一个规则形式

$$A \rightarrow A\alpha \mid \beta$$

其中 α 和 β 代表任意串, 且 β 不以 A 开头。这个规则生成形式 $\beta, \beta\alpha, \beta\alpha\alpha, \beta\alpha\alpha\alpha, \dots$ 的所有串 (所有串均以 β 开头, 其后是零个或多个 α)。因此, 这个文法规则在效果上与正则表达式 $\beta\alpha^*$ 相同。类似地, 右递归文法规则

$$A \rightarrow \alpha A \mid \beta$$

⊖ 这是左递归中的特殊情况, 称作直接左递归 (immediate left recursion), 下一章将讨论一些更普通的情况。

(其中 并不在A处结束)生成所有串 、 、 、 、

如果要编写生成与正则表达式 a^* 相同语言的文法,则文法规则必须有一个用于生成空串的表示法(因为正则表达式 a^* 匹配空串)。这样的文法规则的右边必须为空,可在右边什么也不写,如在

$empty$

中,但大多数情况都使用 ϵ 元符号表示空串(与在正则表达式中的用法类似):

$empty \quad \epsilon$

这样的文法规则称作 ϵ -产生式(ϵ -production)。生成包括了空串的文法必须至少有一个 ϵ -产生式。

现在可以将一个与正则表达式 a^* 相等的文法写作

$A \quad Aa \mid \epsilon$

或

$A \quad aA \mid \epsilon$

两个文法都生成语言 $\{a^n \mid n \text{ 是 } 0 \text{ 的整型}\} = L(a^*)$ 。 ϵ -产生式在定义可选的结构时也很有用,我们马上就能看到这一点。

下面用更多的一些示例来小结这个部分。

例3.5 考虑文法

$A \quad (A)A \mid \epsilon$

这个文法生成所有“配对的括号”的串。例如,串 $((()((()))())$ 就由下面的推导生成(利用 ϵ -产生式去除无用的A):

$A \quad (A)A \quad (A)(A)A \quad (A)(A) \quad (A)() \quad ((A)A)() \\ ((A)A)() \quad ((A)A)A)() \quad ((A)A)() \\ ((A)A)A)() \quad ((A)A)A)() \quad ((A)A)A)()$

例3.6 例3.4中的语句文法用 ϵ -产生式还可写作:

$statement \quad if-stmt \mid other$
 $if-stmt \quad if (exp) statement else-part$
 $else-part \quad else statement \mid \epsilon$
 $exp \quad 0 \mid 1$

请注意, ϵ -产生式指出结构 *else part* 是可选的。

例3.7 考虑一个语句序列的以下文法 G :

$stmt-sequence \quad stmt ; stmt-sequence \mid stmt$
 $stmt \quad s$

这个文法生成由分号分隔开的一个或多个语句序列(语句已被提炼到单个终结符 s 中了):

$L(G) = \{s, s;s, s;s;s, \dots\}$

如要允许语句序列也可为空,则可写出以下的文法 G' :

$stmt-sequence \quad stmt ; stmt-sequence \mid \epsilon$
 $stmt \quad s$

但是它将分号变为一个语句结束符号(terminator)而不是分隔符(separator):

$L(G') = \{\epsilon, s;, s;s;, s;s;s;, \dots\}$

如果允许语句序列可为空，但仍要求保留分号作为语句分隔符，则须将文法写作：

$$\begin{aligned} \text{stmt-sequence} & \quad \text{nonempty-stmt-sequence} \mid \varepsilon \\ \text{nonempty-stmt-sequence} & \quad \text{stmt} \ ; \ \text{nonempty-stmt-sequence} \mid \text{stmt} \\ \text{stmt} & \quad \text{S} \end{aligned}$$

这个例子说明在构造可选结构时，必须留意 ε -产生式的位置。

3.3 分析树与抽象语法树

3.3.1 分析树

推导为构造来自一个初始的非终结符的特定终结符的串提供了一个办法，但是推导并未唯一地表示出它们所构造的结构。总而言之，对于同一个串可有多个推导。例如，使用图 3-1 中的推导从简单表达式文法构造出记号串

$(\text{number} - \text{number}) * \text{number}$

图3-2给出了这个串的另一推导。二者唯一的差别在于提供的替换顺序，而这其实是一个很表面的差别。为了把它表示得更清楚一些，我们需要表示出终结符串的结构，而这些终结符将推导的主要特征抽取出来，同时却将表面的差别按顺序分解开来。这样的表示法就是树结构，它称作分析树。

(1) $\exp \Rightarrow \exp \text{ op } \exp$	$[\exp \rightarrow \exp \text{ op } \exp]$
(2) $\Rightarrow (\exp) \text{ op } \exp$	$[\exp \rightarrow (\exp)]$
(3) $\Rightarrow (\exp \text{ op } \exp) \text{ op } \exp$	$[\exp \rightarrow \exp \text{ op } \exp]$
(4) $\Rightarrow (\text{number} \text{ op } \exp) \text{ op } \exp$	$[\exp \rightarrow \text{number}]$
(5) $\Rightarrow (\text{number} - \exp) \text{ op } \exp$	$[\text{op} \rightarrow -]$
(6) $\Rightarrow (\text{number} - \text{number}) \text{ op } \exp$	$[\exp \rightarrow \text{number}]$
(7) $\Rightarrow (\text{number} - \text{number}) * \exp$	$[\text{op} \rightarrow *]$
(8) $\Rightarrow (\text{number} - \text{number}) * \text{number}$	$[\exp \rightarrow \text{number}]$

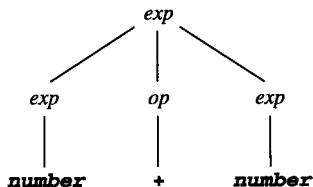
图3-2 表达式 $(34-3)*42$ 的另一个推导

与推导相对应的分析树 (parse tree) 是一个作了标记的树，其中内部的节点由非终结符标出，树叶节点由终结符标出，每个内部节点的子节点都表示推导的一个步骤中的相关非终结符的替换。

以下是一个简单的示例，推导：

$$\begin{aligned} \exp & \quad \exp \text{ op } \exp \\ & \quad \text{number} \text{ op } \exp \\ & \quad \text{number} + \exp \\ & \quad \text{number} + \text{number} \end{aligned}$$

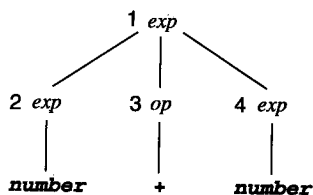
与分析树



相对应。推导中的第 1 步对应于根节点的 3 个孩子。第 2 步对应于根下最左边的 *exp* 的单个 *number* 孩子，后面的两步与上面的类似。通过将分析树中内部节点编号可将这个对应表示得更清楚一些，编号采用在相应的推导中，与其相关的非终结符被取代的步骤编号。因此，如果如下给前一个推导编号：

- (1) *exp* *exp op exp*
- (2) *number* *op exp*
- (3) *number* + *exp*
- (4) *number* + *number*

就可相应地将分析树中的内部节点编号如下：



请注意，该分析树的内部节点的编号实际上是一个前序编号（preorder numbering）。

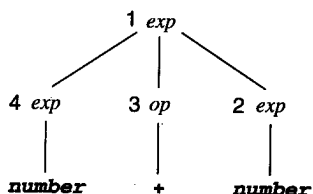
同一个分析树还可与推导

exp *exp op exp*
 exp op number
 exp + number
 number + number

和

exp *exp op exp*
 exp + exp
 number + exp
 number + number

相对应，但是它却提供了内部节点的不同编号。实际上，两个推导中的前一个与下面的编号相对应：



（我们将另一个推导的编号问题留给读者）。此时，该编号与分析树中的内部节点的后序编号（postorder numbering）相反（后序编号将按 4、3、2、1 的顺序访问内部节点）。

一般而言，分析树可与许多推导相对应，所有这些推导都表示与终结符的被分析串相同的基础结构，但是仍有可能找出那个与分析树唯一相关的推导。最左推导（leftmost derivation）是指它的每一步中最左的非终结符都要被替换的推导。相应地，最右推导（rightmost derivation）则是指它的每一步中最右的非终结符都要被替换的推导。最左推导和与其相关的分析树的内部节点的前序编号相对应；而最右推导则和后序编号相对应。

实际上,从刚刚给出的示例中的3个推导和分析中已可看到这个对应了。在3个推导的第1个中给出的是最左推导,而第2个则是最右推导(第3个推导既不是最左推导也不是最右推导)。

再看一个复杂一些的分析树与最左和最右推导的示例——表达式 $(34-3)*42$ 与图3-1和图3-2中的推导。这个表达式的分析树在图3-3中,在其中也根据图3-1的推导为节点编了号。这个推导实际上是最左推导,且分析树的相应编号是相反的后序编号。另一方面,图3-2中的推导是最左推导(我们期望读者能够提供与这个推导相应的分析树的前序编号)。

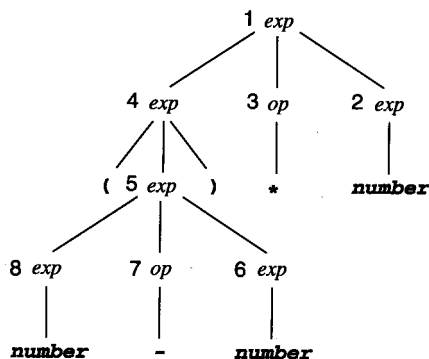
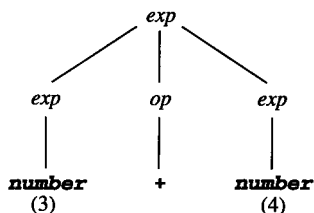


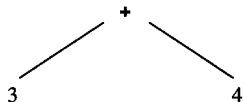
图3-3 算术表达式 $(34-3)*42$ 的分析树

3.3.2 抽象语法树

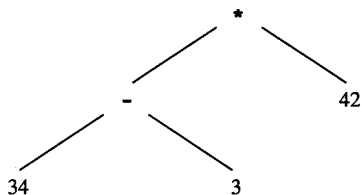
分析树是表述记号串结构的一种十分有用的表示法。在分析树中,记号表现为分析树的树叶(自左至右),而分析树的内部节点则表示推导的各个步骤(按某种顺序)。但是,分析树却包括了比纯粹为编译生成可执行代码所需更多的信息。为了看清这一点,可根据简单的表达式文法,考虑表达式 $3+4$ 的分析树:



这是上一个例子的分析树。我们已经讨论了如何用这个树来显示每一个 **number** 记号的真实数值(这是由扫描程序或分析程序计算的记号的一个特征)。语法引导的转换原则(principle of syntax-directed translation)说明了:如同分析树所表示的一样,串 $3+4$ 的含义或语义应与其语法结构直接相关。在这种情况下,语法引导的转换原则意味着在分析树表示中应加上数值3和数值4。实际上可将这个树看作:根代表两个孩子 **exp** 子树的数值相加。而另一方面,每个子树又代表它的每个 **number** 孩子的值。但是还有一个更为简单的方法能表示与这相同的信息,即如树:



这里的根节点仅是被它所表示的运算标出,而叶子节点由其值标出(不是 **number** 记号)。类似地,在图3-3中给出的表达式 $(34-3)*42$ 的分析树也可由下面的树简单表示出来:



在这个树中，括号记号实际已消失了，但它仍然准确地表达着从 34 中减去 3，然后再乘以 42 的语义内容。

这种树是真正的源代码记号序列的抽象表示。虽然不能从其中重新得到记号序列（不同于分析树），但是它们却包含了转换所需的所有信息，而且比分析树效率更高。这样的树称作抽象语法树（abstract syntax tree）或简称为语法树（syntax tree）。分析程序可通过一个分析树表示所有步骤，但却通常只能构造出一个抽象的语法树（或与它等同的）。

我们可将抽象语法树想象成一个称作抽象语法（abstract syntax）的快速计数法的树形表示法，它很像普通语法结构的分析树表示法（当与抽象语法相比时，也称作具体语法（concrete syntax））。例如，表达式 $3+4$ 的抽象语法可写作 $OpExp(Plus, ConstExp(3), ConstExp(4))$ ；而表达式 $(34-3)*42$ 的抽象语法则可写作：

$$OpExp(Times, OpExp(Minus, ConstExp(34), ConstExp(3)), ConstExp(42))$$

实际上，可通过使用一个类似 BNF 的表示法为抽象语法给出一个正式的定义，这就同具体语法一样。例如，可将简单的算术表达式的抽象语法的相似 BNF 规则写作：

$$\begin{aligned} exp & \quad OpExp(op, exp, exp) \mid ConstExp(integer) \\ op & \quad Plus \mid Minus \mid Times \end{aligned}$$

对此就不再进一步探讨了，我们把主要的精力放在可被分析程序利用的语法树的真正结构上，它由一个数据类型说明给出^①。例如，简单的算术表达式的抽象语法树可由 C 数据类型说明给出：

```
typedef enum {Plus, Minus, Times} OpKind;
typedef enum {OpKind, ConstKind} ExpKind;
typedef struct streenode
{
    ExpKind kind;
    OpKind op;
    struct streenode * lchild, * rchild;
    int val;
} STreeNode;
typedef STreeNode *SyntaxTree;
```

请注意：除了运算本身（加、减和乘）之外，我们在语法树节点两种不同类型（整型常数和运算）中还使用了枚举类型。实际上是可以利用记号来表示运算，而无需定义一个新的枚举类型。此外还能够使用一个 C union 类型来节省空间，这是因为节点不能既是一个算符节点，同时又是一个常数节点。最后还要指出这些树的节点说明只包括了这些示例直接用到的特征。在实际应用中，编译中使用的特征还会更广泛，例如：数据类型、符号表信息等等，本章后面以及以后几章的示例都会谈到它。

下面用一些通过使用前面例子中谈到过的文法的分析树和语法树的示例来小结这一段。

例 3.8 考虑例 3.4 中被简化了的 if 语句的文法：

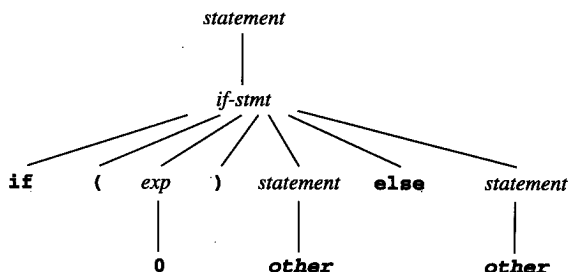
$$\begin{aligned} statement & \quad if-stmt \mid other \\ if-stmt & \quad if (exp) statement \\ & \quad \mid if (exp) statement else statement \\ exp & \quad 0 \mid 1 \end{aligned}$$

串

^① 有一些刚刚给出的抽象语法的语言在本质上是一个类型说明，参见练习。

if (0) other else other

的分析树是：

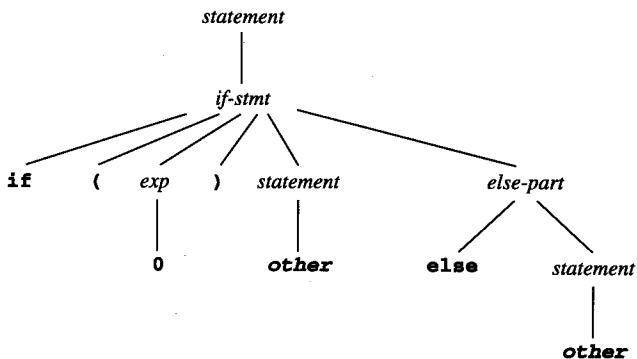


利用例3.6中的文法

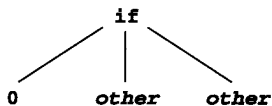
```

statement  if-stmt | other
if-stmt   if (exp) statement else-part
else-part else statement | ε
exp       0 | 1
  
```

这个串有以下的分析树：



if语句除了3个附属结构之外，它就什么也不需要了，这3个附属结构分别是：测试表达式、then-部分和else部分（如果出现），因此前面一个串的语法树（利用例3.4或例3.6中的文法）是



在这里我们将保留的记号 **if** 和 **other** 用作标记以区分语法树中的语句类型。利用枚举类型则会更为恰当一些。例如，利用一个C声明的集合将本例的语句结构和表达式相应地表示如下：

```

typedef enum { ExpK, StmtK } NodeKind;
typedef enum { Zero, One } ExpKind;
typedef enum { IfK, OtherK } StmtKind;
typedef struct streenode
{
    NodeKind kind;
    ExpKind ekind;
    StmtKind skind;
    struct streenode
        *test, *thenpart, *elsepart;
}
  
```



```

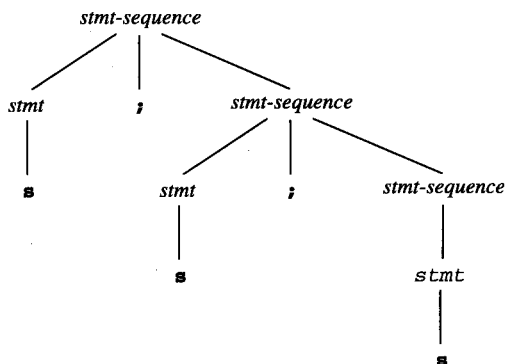
} STreeNode;
typedef STreeNode * SyntaxTree;

```

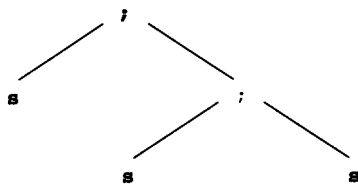
例3.9 考虑将例3.7中的语句序列的文法用分号分隔开：

$stmt_sequence \quad stmt ; stmt_sequence \mid stmt$
 $stmt \quad s$

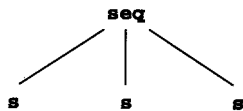
这个 $s; s; s$ 串有关于这个文法的以下分析树：



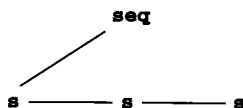
这个串可能有一个语法树：



除了它们的“运算”仅仅是将一个序列中的语句绑定在一起之外，这个树中的分号节点与算符节点（如算术表达式中的 + 节点）类似。我们可以尝试着将一个序列中的所有语句节点都与一个节点绑定在一起，这样前面的语法树就变成



但它存在一个问题： seq 节点可以有任意数目的孩子，而又很难在一个数据类型说明中提供它。其解决方法是利用树的标准最左孩子右同属（leftmost-child right-sibling）来表示（在大多数数据结构文本中都有）。在这种表示中，由父亲到它的孩子的唯一物理连接是到最左孩子的。孩子则在一个标准连接表中自左向右连接到一起，这种连接称作同属（sibling）连接，用于区别父子连接。在最左孩子右同属安排下，前一个树现在就变成了：



有了这个安排，我们还可以去掉连接的 seq 节点，那么语法树变得更简单了：



这很明显是用于表示语法树中一个序列的最简单和最方便的方法了。其复杂之处在于这里的连

接是同属连接，它必须与子连接相区分，而这又要求在语法树说明中有一个新的域。

3.4 二义性

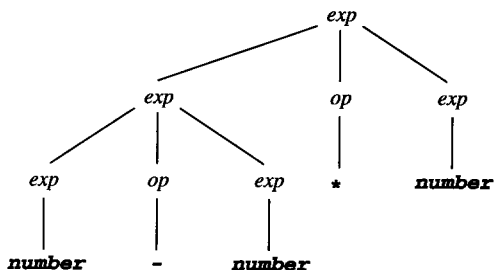
3.4.1 二义性文法

分析树和语法树唯一地表达着语法结构，它们与表达最左和最右推导一样，但并不是对于所有推导都可以。不幸的是，文法有可能允许一个串有多于一个的分析树。例如在前面作为标准示例的简单整型算术文法中

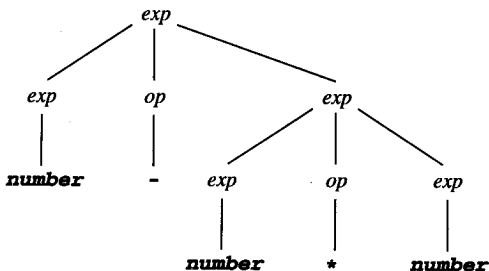
$$\text{exp} \quad \text{exp op exp} \mid (\text{exp}) \mid \text{number}$$

$$\text{op} \quad + \mid - \mid *$$

和串 $34-3*42$ ，这个串有两个不同的分析树：



和



它们与两个最左推导相对应：

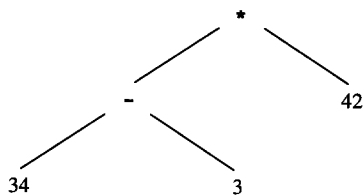
$\text{exp} \Rightarrow \text{exp op exp}$	$[\text{exp} \rightarrow \text{exp op exp}]$
$\Rightarrow \text{exp op exp op exp}$	$[\text{exp} \rightarrow \text{exp op exp}]$
$\Rightarrow \text{number op exp op exp}$	$[\text{exp} \rightarrow \text{number}]$
$\Rightarrow \text{number} - \text{exp op exp}$	$[\text{op} \rightarrow -]$
$\Rightarrow \text{number} - \text{number op exp}$	$[\text{exp} \rightarrow \text{number}]$
$\Rightarrow \text{number} - \text{number} * \text{exp}$	$[\text{op} \rightarrow *]$
$\Rightarrow \text{number} - \text{number} * \text{number}$	$[\text{exp} \rightarrow \text{number}]$

和

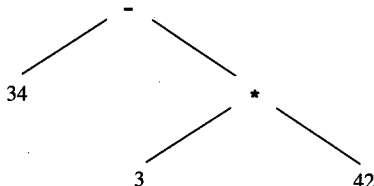
$\text{exp} \Rightarrow \text{exp op exp}$	$[\text{exp} \rightarrow \text{exp op exp}]$
$\Rightarrow \text{number op exp}$	$[\text{exp} \rightarrow \text{number}]$
$\Rightarrow \text{number} - \text{exp}$	$[\text{op} \rightarrow -]$
$\Rightarrow \text{number} - \text{exp op exp}$	$[\text{exp} \rightarrow \text{exp op exp}]$
$\Rightarrow \text{number} - \text{number op exp}$	$[\text{exp} \rightarrow \text{number}]$

$\Rightarrow \text{number} - \text{number} * \text{exp}$ $[op \rightarrow *]$
 $\Rightarrow \text{number} - \text{number} * \text{number}$ $[exp \rightarrow \text{number}]$

则相应的语法树为：



和



可生成带有两个不同分析树的串的文法称作二义性文法 (ambiguous grammar)。由于这个文法并不能准确地指出程序的语法结构 (即使是完全确定正规串本身 (它是文法的语言成员)), 所以它是分析程序表示的一个严重问题。在某种意义上, 二义性文法就像是一个非确定的自动机, 此时两个不同的路径都可接收相同的串。但是因为没有一个合适的算法, 因此就不能像自动机中的情形一样 (第2章中讨论过的子集构造), 文法中的二义性就不能如有穷自动机中的非确定性一样轻易地被解决^①。

所以必须将二义性文法认为是一种语言语法的不完善说明, 而且也应避免它。幸运的是, 二义性文法在后面将介绍到的标准分析算法的测试中总是失败的, 而且也开发出了标准技术体系来解决在程序设计语言中遇到的典型二义性。

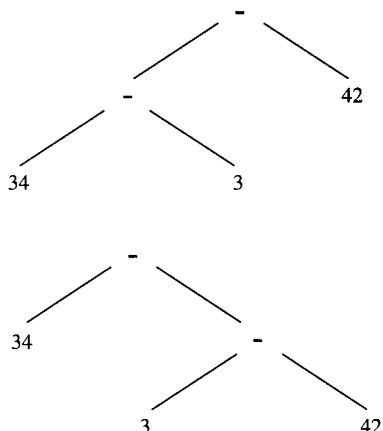
有两个解决二义性的基本方法。其一是: 设置一个规则, 该规则可在每个二义性情况下指出哪一个分析树 (或语法树) 是正确的。这样的规则称作消除二义性规则 (disambiguating rule)。这样的规则的用处在于: 它无需修改文法 (可能会很复杂) 就可消除二义性; 它的缺点在于语言的语法结构再也不能由文法单独提供了。另一种方法是将文法改变成一个强制正确分析树的构造的格式, 这样就可以解决二义性了。当然在这两种办法中, 都必须确定在二义性情况下哪一个树是正确的。这就再一次涉及到语法制导翻译原则了。我们所需的分析 (或语法) 树应能够正确地反映将来应用到构造的意义, 以便将其翻译成目标代码。

在前面的两个语法树中, 哪一个串 34-3*42 的正确解释呢? 第1个树通过把减法节点作为乘法节点的孩子, 指出可将这个表达式等于: 先做减法 (34-3=31), 然后再做乘法 (31*42=1302)。相反地, 第2个树指出先做乘法 (3*42=126) 然后再做减法 (34-126=-92)。选择哪个树取决于我们认为哪一个计算是正确的。人们认为乘法比减法优先 (precedence)。通常地, 乘法和除法比加法和减法优先。

为了去除在这个简单表达式文法中的二义性, 现在可以只需设置消除二义性规则, 它建立了3个运算相互之间的优先关系。其标准解决办法是给予加法和减法相同的优先权, 而乘法和除法则有高一级的优先权。

^① 情况甚至更糟, 这是因为没有算法可以在一开始时就确定一个文法是否是二义性的, 参见第3.2.7节。

不幸的是，这个规则仍然不能完全地去除掉文法中的二义性。例如串 $34-3-42$ ，这个串也有两种可能的语法树：



和

第1个语法树表示计算 $(34-3)-42=-11$ ，而第2个表示计算 $34-(3-42)=73$ 。而哪一个算式正确又是一个惯例的问题，标准数学规定第1种选择是正确的。这是由于规定减法为左结合（left associative），也就是认为一个减法序列的运算是自左向右的。

因此，这又要求有一个消除二义性的规则：它能处理每一个加法、减法和乘法的结合性。这个消除二义性的规则通常都规定它们为左结合，它确实消除了简单表达式文法中其他的二义性问题（在后面再证明它）。

因为在表达式中不允许有超过一个算符的序列，有时还需要规定运算是非结合性的（nonassociative）。例如，可将简单表达式文法用下面的格式写出：

```

exp    factor op factor | factor
factor ( exp ) | number
op     + | - | *
  
```

在这种情况下，诸如 $34-3-42$ 甚至于 $34-3*42$ 的表达式都是正规的，但它们必须带上括号，例如 $(34-3)-42$ 和 $34-(3*42)$ 。这样的完全括号表达式（fully parenthesized expression）就没有必要说明其结合性或优先权了。上面的文法正如所写的一样去除了二义性。当然，我们不仅改变了文法，还更改了正被识别的语言。

我们不再讨论消除二义性的规则，现在来谈谈重写文法以消除二义性的方法。请注意，必须找到无需改变正被识别的基本串的办法（正如完全括号表达式的示例所做的一样）。

3.4.2 优先权和结合性

为了处理文法中的运算优先权问题，就必须把具有相同优先权的算符归纳在一组中，并为每一种优先权规定不同的规则。例如，可把乘法比加法和减法优先添加到简单表达式文法，如下所示：

```

exp    exp addop exp | term
addop   + | -
term    term mulop term | factor
mulop   *
factor  ( exp ) | number
  
```

在这个文法中，乘法被归在 *term* 规则下，而加法和减法则被归在 *exp* 规则之下。由于 *exp* 的

基本情况是 *term*，这就意味着加法和减法在分析树和语法树中将被表现地“更高一些”(也就是，更接近于根)，由此也就接受了更低一级的优先权。这样将算符放在不同的优先权级别中的办法是在语法说明中使用BNF的一个标准方法。这种分组称作优先级联 (precedence cascade)。

简单算术表达式的最后一种文法仍未指出算符的结合性而且仍有二义性。它的原因在于算符两边的递归都允许每一边匹配推导 (因此也在分析树和语法树) 中的算符重复。解决方法是用基本情况代替递归，强制重复算符匹配一边的递归。这样将规则

$$\text{exp} \quad \text{exp addop exp} \mid \text{term}$$

替换为

$$\text{exp} \quad \text{exp addop term} \mid \text{term}$$

使得加法和减法左结合，而

$$\text{exp} \quad \text{term addop exp} \mid \text{term}$$

却使得它们右结合。换言之，左递归规则使得它的算符在左边结合，而右递归规则使得它们在右边结合。

为了消除简单算术表达式BNF规则中的二义性，重写规则使得所有的运算都左结合：

$$\text{exp} \quad \text{exp addop term} \mid \text{term}$$

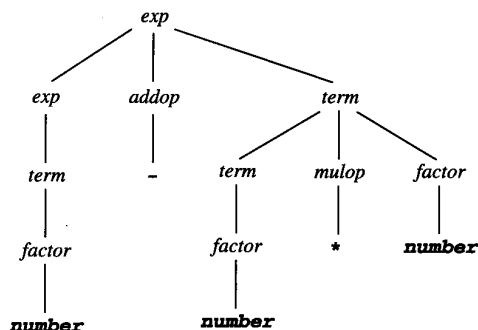
$$\text{addop} \quad + \mid -$$

$$\text{term} \quad \text{term mulop factor} \mid \text{factor}$$

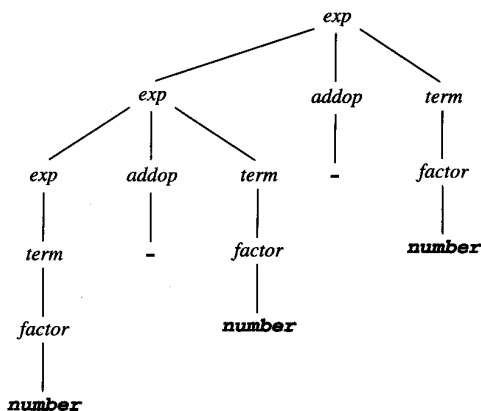
$$\text{mulop} \quad *$$

$$\text{factor} \quad (\text{exp}) \mid \text{number}$$

现在表达式 $34-3*42$ 的分析树就是



表达式 $34-3*42$ 的分析树是：



注意，优先级联使得分析树更为复杂。但是语法树并不受影响。

3.4.3 悬挂else问题

考虑例3.4中的文法：

```

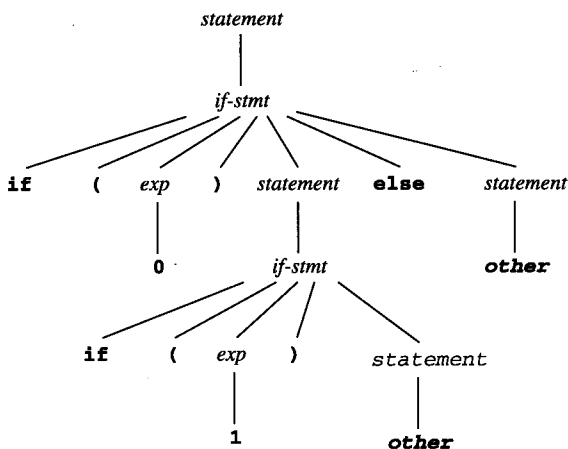
statement  if-stmt | other
if-stmt    if( exp ) statement
           | if( exp ) statement else statement
exp        0 | 1

```

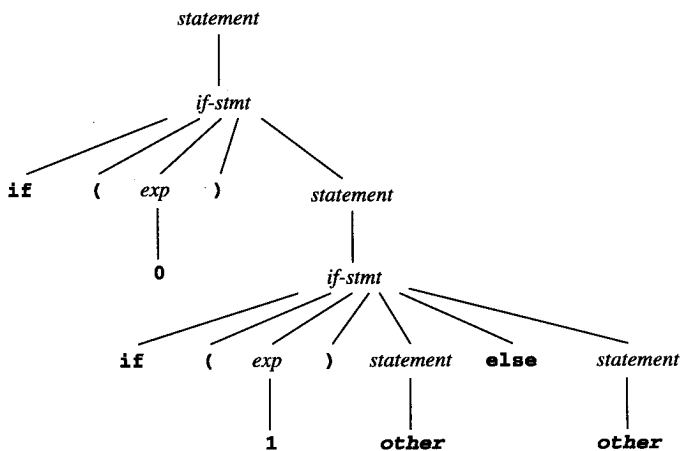
由于可选的else的影响，这个文法有二义性。为了看清它，可考虑下面的串：

if (0) if (1) other else other

这个串有两个分析树：



和



哪一个是正确的则取决于将单个 else 部分与第 1 个或第 2 个 if 语句结合：第 1 个分析树将 else 部分与第 1 个 if 语句结合；第 2 个分析树将它与第 2 个 if 语句结合。这种二义性称作悬挂 else 问题 (dangling else problem)。为了分清哪一个分析树是正确的，我们必须考虑 if 语句的隐含意思，请看下面的一段 C 代码：


```

if (x != 0)
    if (y == 1/x) ok = TRUE
    else z = 1/x

```

在这个代码中，如果else部分与第1个if语句结合，只要x等于0，则会发生一个除以零的错误。因此这个代码的含义（实际是else部分缩排的含义）是指一个else部分应总是与没有else部分的最近的if语句结合。这个消除二义性的规则被称为用于悬挂else问题的最近嵌套规则（most closely nested rule），这就意味着上面第2个分析树是正确的。请注意：如要将else部分与第1个if语句相结合，则要用C中的{...}，如在

```

if (x != 0)
    {if (y == 1/x) ok = TRUE;}
else z = 1/x;

```

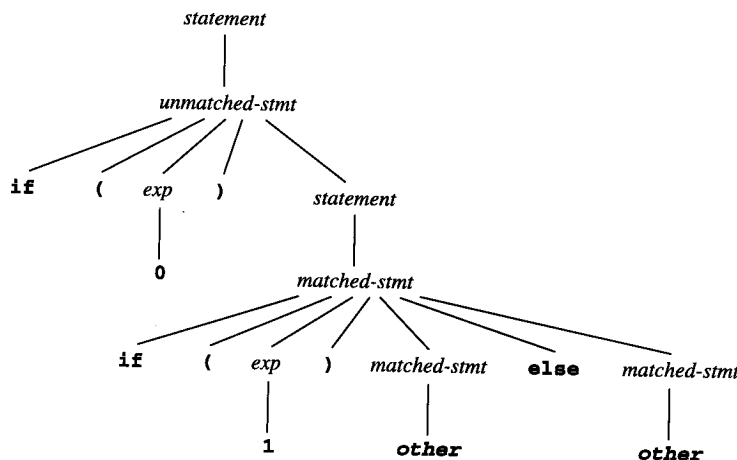
解决这个位于BNF本身中的悬挂else二义性要比处理前面的二义性困难。方法如下：

```

statement    matched-stmt | unmatched-stmt
matched-stmt  if( exp ) matched-stmt else matched-stmt | other
unmatched-stmt if( exp ) statement
               | if( exp ) matched-stmt else unmatched-stmt
exp           0 | 1

```

它允许在if语句中，只有一个matched-stmt出现在else之前，这样就迫使尽可能快地匹配所有的else部分。例如，经过结合了的简单串的分析树现在就变成了：



它确实将else部分与第2个if语句相连。

通常不能在BNF中建立最近嵌套规则，实际上经常所用的是消除二义性的规则。原因之一是它增加了新文法的复杂性，但主要原因却是分析办法很容易按照遵循最近嵌套规则的方法来配置（在无需重写文法时自动获得优先权和结合性有一点困难）。

悬挂else问题来源于Algol60的语法。我们还是有可能按照悬挂else问题不出现的方法来设计语法。办法之一是要求出现else部分，而该办法已在LISP和其他函数语言中用到了（但须返回一个值）。另一种办法是为if语句使用一个带括号关键字（bracketing keyword）。使用这种方法的语言包括了Algol68和Ada。例如，程序员写

```

if x /=0 then
    if y=1/x then ok := true ;

```

```

    else z := 1/x ;
  end if;
end if;

```

将else部分与第2个if语句结合在一起。程序员还可写出

```

if x /=0 then
  if y = 1/x then ok := true;
  end if;
else z := 1/x;
end if;

```

将它与第1个if语句结合在一起。Ada中相应的BNF（有一些简单化了）就是

```

if-stmt  if condition then statement-sequence end if
        | if condition then statement-sequence
          else statement-sequence end if

```

因此两个关键字end if就是Ada的括号关键字。在Algol68中，括号关键字是fi（反着写的if）。

3.4.4 无关紧要的二义性

有时文法可能会有二义性并且总是生成唯一的抽象语法树。例如，在例 3.9中的语句序列文法中，可选择类似于语法树的简单同属表。在这种情形下，右递归文法规则或左递归文法规则仍导致相同的语法树结构，且可将文法二义地写作

```

stmt-sequence  stmt-sequence ; stmt-sequence | stmt
stmt          s

```

且仍然可得到唯一的语法树。由于相结合的语义不必依赖于使用的是哪种消除二义性的规则，所以可将这样的二义性称作无关紧要的二义性（inessential ambiguity）。同样的情况出现在二进制算符中，例如算术加法或串的并置表现为可结合运算（associative operation）（若对于所有的值 a 、 b 和 c ，且有 $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ ，那么二进制算符“ \cdot ”也是可结合的）。此时的语法树仍然各不相同，但是却表示相同的语义值，而且我们可能也无需在意到底使用的是哪一个。然而，分析算法却要提供一些消除二义性的规则，这些都是编译器编写者所需的。

3.5 扩展的表示法：EBNF和语法图

这一部分将对两种扩展表示法分别进行讲解。

3.5.1 EBNF表示法

重复和可选的结构在程序设计语言中极为普通，因此在BNF文法规则中也是一样的。所以当看到BNF表示法有时扩展到包括了用于这两个情况的特殊表示法时，也不应感到惊奇。这些扩展包含了称作扩展的BNF（extended BNF）或EBNF的表示法。

首先考虑重复情况，如在语句序列中。我们已看到重复是由文法规则中的递归表达，并可能使用了左递归或右递归，它们由一般规则

$A \quad A \mid$ （左递归）

和

$A \quad A \mid$ （右递归）

指出, 其中 ϵ 和 $\$$ 是终结符和非终结符的任意串, 且在第 1 个规则中 ϵ 不以 A 开始, 在第 2 个规则中 $\$$ 不以 A 结束。

重复有可能使用与正则表达式所用相同的表示法, 即: 星号 $*$ (在正则表达式中也称作 Kleene 闭包)。则这两个规则可被写作非递归规则

$$A \rightarrow \epsilon \mid A^*$$

和

$$A \rightarrow A^*$$

相反地, EBNF 选择使用花括号 $\{ \dots \}$ 来表示重复 (因此清晰地表达出被重复的串的范围), 且可为规则写出

$$A \rightarrow \{ A \}$$

和

$$A \rightarrow \{ A \}$$

使用重复表示法的问题是: 它使得分析树的构造不清楚, 但是正如所看到的一样, 我们对此并不在意, 例如语句序列的情形 (例 3.9)。在右递归格式中写出如下文法:

$$\begin{aligned} \text{stmt-sequence} &\rightarrow \text{stmt} ; \text{stmt-sequence} \mid \text{stmt} \\ \text{stmt} &\rightarrow S \end{aligned}$$

这个规则具有格式 $A \rightarrow A \mid \epsilon$, 且 $A = \text{stmt-sequence}$, $\epsilon = \text{stmt} ; \text{stmt}$ 。在 EBNF 中, 它表现为:

$$\text{stmt-sequence} \rightarrow \{ \text{stmt} ; \} \text{stmt} \quad (\text{右递归格式})$$

同样也可使用一个左递归规则并得到 EBNF

$$\text{stmt-sequence} \rightarrow \text{stmt} \{ ; \text{stmt} \} \quad (\text{左递归格式})$$

实际上, 第 2 个格式是通常所用的 (原因在下一章再讲)。

出现在结合性中的更大的一个问题是发生在诸如二进制运算的减法和除法中。例如, 前面减法的简单表达式文法中的第 1 个文法规则:

$$\text{exp} \rightarrow \text{exp} \text{ addop } \text{term} \mid \text{term}$$

它使得 $A \rightarrow A \mid \epsilon$, 且 $A = \text{exp}$, $\epsilon = \text{addop term}$, $\epsilon = \text{term}$ 。因此, 将这个规则在 EBNF 中写作:

$$\text{exp} \rightarrow \text{term} \{ \text{addop term} \}$$

尽管规则本身并未明显地说明出来, 但现在仍可假设它暗示了左结合。我们还可假设通过写出

$$\text{exp} \rightarrow \{ \text{term addop} \} \text{term}$$

来暗示一个右结合规则, 但事实并不是这样。相反地, 诸如

$$\text{stmt-sequence} \rightarrow \text{stmt} ; \text{stmt-sequence} \mid \text{stmt}$$

的右递归规则可被看作后接一个可选的分号和 stmt-sequence 的 stmt 。

EBNF 中的可选结构可通过前后用方括号 $[\dots]$ 表示出来。这同将问号放在可选部分之后的正则表达式惯例本质上是一样的, 但它另有无需括号就可将可选部分围起来的优点。例如, 用带有可选的 `else` 部分的 `if` 语句 (例 3.4 和例 3.6) 的文法规则在 EBNF 中写作:

$$\begin{aligned} \text{statement} &\rightarrow \text{if-stmt} \mid \text{other} \\ \text{if-stmt} &\rightarrow \text{if} (\text{exp}) \text{statement} [\text{else statement}] \end{aligned}$$

$$exp \quad 0 \mid 1$$

而诸如

$$stmt\text{-}sequence \quad stmt \ ; \ stmt\text{-}sequence \mid stmt$$

的右递归可写作：

$$stmt\text{-}sequence \quad stmt \ [\ ; \ stmt\text{-}sequence \]$$

(请与前面使用花括号写在左递归格式中的这个规则作一比较)。

如果希望在右结合中写出一个诸如加法的算术运算，则可用

$$exp \quad term \ [\ addop \ exp \]$$

来代替花括号的使用。

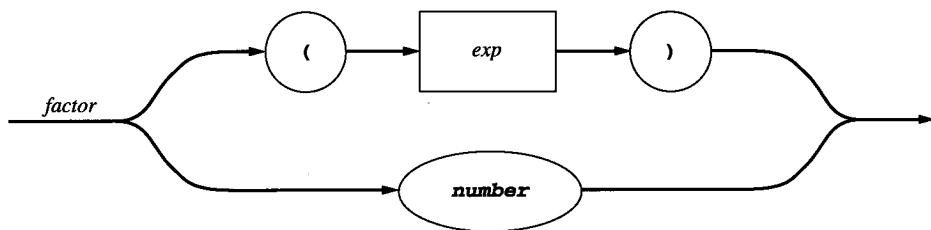
3.5.2 语法图

用作可视地表示EBNF规则的图形表示法称作语法图 (syntax diagram)。它们是由表示终结符和非终结符的方框、表示序列和选择的带箭头的线，以及每一个表示文法规则定义该非终结符的图表的非终结符标记记组成。圆形框和椭圆形框用来指出图中的终结符，而方形框和矩形框则用来指出非终结符。

例如，文法规则

$$factor \quad (\ exp \) \mid \mathbf{number}$$

用语法图表示则是：

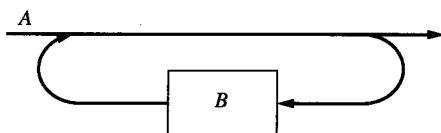


请注意，*factor* 并未放在框中，而是用作语法图的标记来指出该图表示该名称结构的定义。另外还需注意带有箭头的线用作指明选择和顺序。

语法图是从EBNF而非BNF中写出来的，所以需要用图来表示重复和可选结构。给出下面的重复：

$$A \ \{ \ B \}$$

相对应的语法图通常画作

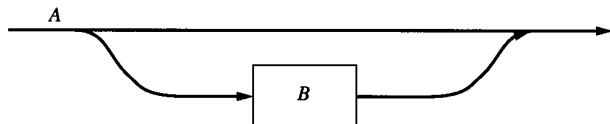


请留意该图必须允许根本没有出现 *B*。

诸如

$$A \ \[\ B \]$$

的可选结构可画作



下面的示例是几个使用了前面EBNF的例子，我们用它们来小结语法图的讨论。

例3.10 考虑简单算术表达式的运行示例。它具有BNF（包括结合性和优先权）。

```

exp    exp addop term | term
addop  + | -
term   term mulop factor | factor
mulop  *
factor ( exp ) | number

```

相对应的EBNF是

```

exp    term { addop term }
addop  + | -
term   factor { mulop factor }
mulop  *
factor ( exp ) | number

```

图3-4是相对应的语法图（factor的语法图已在前面给出了）。

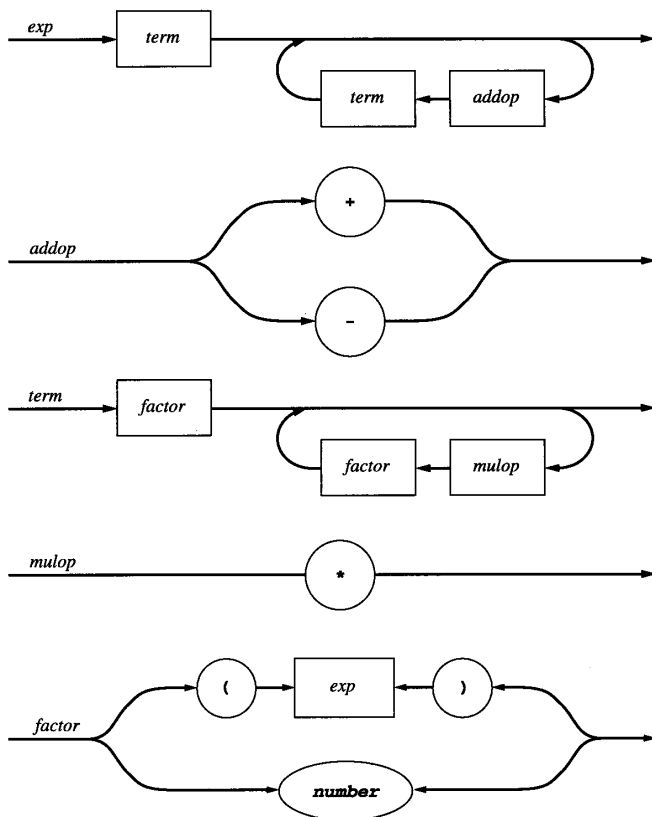


图3-4 例3.10中文法的语法图

例3.11 考虑例3.4中简化了的if语句的文法。它有BNF

$$\begin{aligned} \text{statement} & \quad \text{if-stmt} \mid \text{other} \\ \text{if-stmt} & \quad \text{if (exp) statement} \\ & \quad \mid \text{if (exp) statement else statement} \\ \text{exp} & \quad 0 \mid 1 \end{aligned}$$

EBNF为

$$\begin{aligned} \text{statement} & \quad \text{if-stmt} \mid \text{other} \\ \text{if-stmt} & \quad \text{if (exp) statement [else statement]} \\ \text{exp} & \quad 0 \mid 1 \end{aligned}$$

图3-5是相对应的语法图。

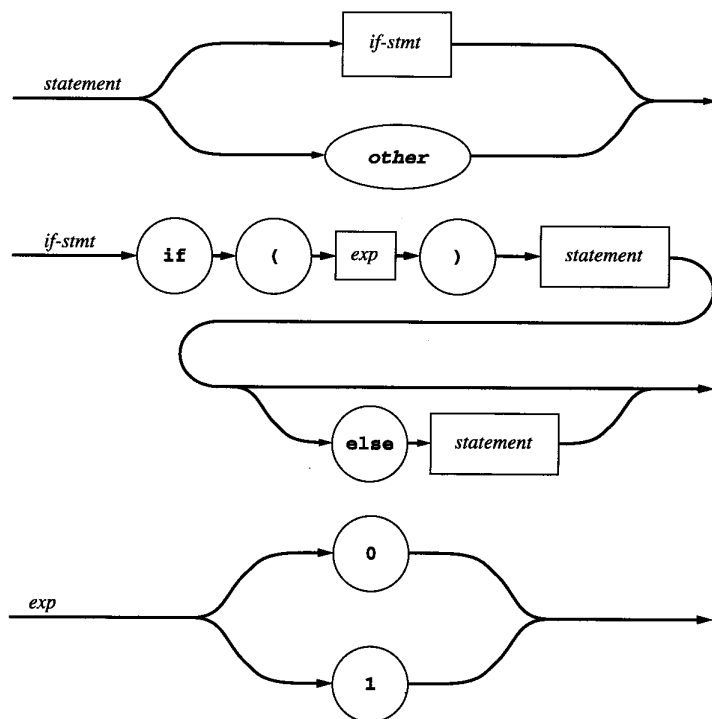


图3-5 例3.11中文法的语法图

3.6 上下文无关语言的形式特性

在这里用更正式的数学方法写出本章前面所提到过的一些术语和定义。

3.6.1 上下文无关语言的形式定义

本节给出上下文无关文法的形式定义。

定义 上下文无关文法由以下各项组成：

- 1) 终结符 (terminal) 集合 T 。
- 2) 非终结符 (nonterminal) 集合 N (与 T 不相交)。

3) 产生式 (production) 或文法规则 (grammar rule) A 的集合 P , 其中 A 是 N 的一个元素, ϵ 是 $(T \cup N)^*$ 中的一个元素 (是终结符和非终结符的一个可为空的序列)。

4) 来自集合 N 的开始符号 (start symbol) S 。

令 G 是一个如上所定义的文法, 则 $G = (T, N, P, S)$ 。 G 上的推导步骤 (derivation step) 格式 $A \rightarrow B$, 其中 A 和 B 是 $(T \cup N)^*$ 中的元素, 且有 A 在 P 中 (终结符和非终结符的并集, 有时被称作 G 的符号集, set of symbol), 且 $(T \cup N)^*$ 中的串 B 被称作句型 (sentential form)。将关系 \Rightarrow 定义为推导步骤关系 \rightarrow 的传递闭包; 也就是: 假若有零个或多个推导步骤, 就有 $A \Rightarrow B$ ($n \geq 0$)

其中 $A = a_1 a_2 \dots a_{n-1} a_n$ (如果 $n=0$, 则 $A = \epsilon$)。在文法 G 上的推导 (derivation) 形如 $S \Rightarrow^* w$, 且 $w \in T^*$ (即: w 是终结符的一个串, 称作句子 (sentence)), S 是 G 的开始符号。

由 G 生成的语言 (language generated by G) 写作 $L(G)$, 它被定义为集合 $L(G) = \{ w \in T^* \mid \text{存在 } G \text{ 的一个推导 } S \Rightarrow^* w \}$, 也就是: $L(G)$ 是由 S 推导出的句子的集合。

最左推导 (leftmost derivation) $S \Rightarrow^* w$ 是一个推导, 在其中的每一个推导步骤 $A \rightarrow B$ 都有 $A = T^* A'$, 换言之, A' 仅由终结符组成。类似地, 最右推导 (rightmost derivation) 就是每一个推导步骤 $A \rightarrow B$ 都有属性 $A = A' T^*$ 。

文法 G 上的分析树 (parse tree) 是一个带有以下属性的作了标记的树:

- 1) 每个节点都用终结符、非终结符或 ϵ 标出。
- 2) 根节点用开始符号 S 标出。
- 3) 每个叶节点都用终结符或 ϵ 标出。
- 4) 每个非叶节点都用非终结符标出。
- 5) 如带有标记 $A \in N$ 的节点有 n 个带有标记 X_1, X_2, \dots, X_n 的孩子 (可以是终结符也可以是非终结符), 就有 $A \rightarrow X_1 X_2 \dots X_n \in P$ (文法的一个产生式)。

每一个推导都引出一个分析树, 这个分析树中的每一个步骤 $A \rightarrow B$ 都在推导中, 且 $B = X_1 X_2 \dots X_n$ 与带有标记 X_1, X_2, \dots, X_n 的 n 个孩子的结构相对应, 其中 X_1, X_2, \dots, X_n 带有标记 A 。许多推导可引出相同的分析树。但每个分析树只有唯一的一个最左推导和一个最右推导。最左推导与分析树的前序编号相对应, 而与之相反, 最右推导与分析树的后序编号相对应。

若上下文无关文法 G 有 $L = L(G)$, 就将串 L 的集合称作上下文无关语言 (context-free language)。一般地, 许多不同的文法可以生成相同的上下文无关语言, 但是根据所使用的文法不同, 语言中的串也会有不同的分析树。

若存在串 $w \in L(G)$, 其中 w 有两个不同的分析树 (或最左推导或最右推导), 那么文法 G 就有二义性 (ambiguous)。

3.6.2 文法规则和等式

在本节的开始, 我们注意到文法规则用箭头符号而不是等号来定义结构名称 (非终结符), 这与在正则表达式中用等号定义名称的表示法不同。其原因在于文法规则的递归本质使得定义关系 (文法规则) 并不完全与相等一样, 而且我们确实也看到了从推导得出的由文法规则定义的串, 而且在BNF中的箭头指示之后, 使用了一个从左到右的替换方法。

但文法规则的左、右边仍然保存某种等式关系, 但由这个观点引出的语言定义过程与正则表达式中的不同。这个观点对于程序设计语言语义理论很重要, 且由于它对诸如分析的递归

过程的重要作用（即使我们所学习的分析算法并不基于此），因此值得学习。

例如下面的文法规则是从简单表达式文法中抽取出来的（在简化了的格式中）：

$$exp \quad exp + exp \mid number$$

我们已经看到如 exp 的非终结符名称定义了一个终结符的串集合（若非终结符是开始符号，那么它就是文法的语言）。假设将这个集合称作 E ，并令 N 为自然数集（与正则表达式名称 $number$ 对应），则给出的文法规则可解释为集合等式：

$$E = (E + E) \quad N$$

其中 $E+E$ 是串 $\{u + v \mid u, v \in E\}$ 的集合（这里并未添加串 u 和串 v ，而只是用符号“+”将它们连接在一起）。

这是集合 E 的递归等式。思考一下它是如何定义 E 的。首先，由于 E 是 N 与 $E+E$ 的并集，所以 E 中包含了集合 N （这是基本情况）。其次，等式 E 中也包含了 $E+E$ ，就意味着 $N+N$ 也在 E 中。而且又由于 N 和 $N+N$ 均在 E 中，所以 $N+N+N$ 也在 E 中，同理类推。我们可以把这看作是一个更长的串的归纳结构，且所有这些集合的和就是所需结果：

$$E = N \quad (N + N) \quad (N + N + N) \quad (N + N + N + N) \quad \dots$$

实际上，可以证明这个 E 满足正在讨论的等式，而 E 其实是最小的集合。如将 E 的等式右边看作是 E 的一个函数（集），则可定义出 $f(s) = (s + s) \quad N$ ，此时 E 的等式就变成了 $E = f(E)$ 。换言之， E 是函数 f 的一个固定点（fixed point），且（根据前面的注释）它确实是这样一个最小的点。我们将由这种办法定义的 E 看作是给定的最小的固定点语义（least-fixed point semantics）。

当根据本书后面要讲到的通用算法来完成它们时，被递归地定义的程序设计语言，例如语法、递归数据类型和递归函数，都能证明出它具有最小的固定点语义。因为以后将会用到这样的办法来证明编译的正确性，所以这十分重要。现在的编译器很少能被证明是正确的。而编译器代码的测试只能证实近似地正确，而且仍然保留了许多错误，即使是商品化生产的编译器也是这样的。

3.6.3 乔姆斯基层次和作为上下文无关规则的语法规限

对于表示程序设计语言的语法结构，BNF和EBNF中的上下文无关文法是一个有用且十分高效的工具，但是掌握BNF能够和应该表示什么同样也十分重要。我们已经遇到过故意使文法具有二义性的情形（悬挂else问题），那么因此也就不能直接表示出完整的语法来了。当试图在文法中表示太多的东西时，或在文法中表示一个极不可能的要求时，会出现其他情况。本节将讨论一些常见的情况。

当为某个语言编写BNF时，会遇到一个常见的问题，那就是：词法结构的范围应表示在BNF中而不是在一个单独的描述中（可能使用正则表达式）。前面的讨论已指出上下文无关文法可以表达并置、重复和选择，这与正则表达式是一样的。因此我们能够对所有来自字符的记号结构写出文法规则，并将其与正则表达式一起分配。

例如，考虑一个使用正则表达式定义作为数字序列的数：

```
digit = 0|1|2|3|4|5|6|7|8|9
number = digit digit*
```

用BNF写出这个定义：

```
digit 0|1|2|3|4|5|6|7|8|9
number number digit|digit
```

注意，第2个规则中的递归仅仅是用来表达重复。人们认为带有这个属性的文法是一个正规文法（regular grammar），正规文法能够表达任何正则表达式可以表达的内容。因此，我们就可以设计一个分析程序，这个程序可以直接从输入源文件中接收字符并与扫描程序一起分配。

它确实是一个很好的办法。分析程序是一个比扫描程序更有效的机器，但相应地效率要差一些。然而我们还是有理由在BNF本身中包括记号的定义，文法会接着表达完整的语法结构，其中还包括了词法结构。当然，人们希望语言设备能从文法中抽取这些定义并将它们返回到一个扫描程序之上。

另一种情况的发生与上下文规则（context rule）有关，它经常发生在程序设计语言中。我们以前一直使用着术语“上下文无关”，但实际上却并未解释为什么这样的规则是“上下文无关”。其简单原因在于非终结符本身就出现在上下文无关规则中箭头的左边。因此，规则

$$A$$

就说明了无论在何处发生 A ，它都可被任何地方的 α 所替代。另一方面，可以将上下文（context）非正式地定义为（终结符和非终结符的）一对串 α 和 β ，这样仅当 α 和 β 分别在非终结符的前面和后面发生时，才提供一个规则。可将此写作

$$A$$

当 ϵ 时，该规则称作上下文有关文法规则（context-sensitive grammar rule）。上下文有关文法规则比上下文无关文法规则更强大，但要作为分析程序的基础却仍有许多困难。

在程序设计语言中，对于上下文有关规则有哪些要求呢？典型的要求包括了名称的使用。要求在使用之前先说明（declaration before use）的C规则就是一个典型的例子。此时在允许语句或表达式使用名称之前必须在一个说明中先出现一个名称：

```
{ int x;
...
...x...
...
}
```

如果想用BNF规则处理这个要求，也是可以做到的。首先，文法规则必须包括了名称串本身，而不是包括不可区别的作为标识符记号的所有名称。其次，须为每个名称写出一个建立在可能的使用之前的说明规则。但是在许多语言中，标识符的长度是非限制性的，因此标识符的数量也是（至少是可能的）无限的。即使只允许名称为两个字符，仍然有可能有几百种新的文法规则。这很明显是不实际的。这种情形与消除二义性规则的情况类似：只需说出在文法中不是显式的规则（使用之前的声明）。但两者有区别：这样的规则不能作为分析程序本身，这是由于它超出了（可能的）上下文无关规则所能表达的能力。相反地，由于它取决于符号表的使用（它记录了声明的是哪一些标识符），因此该规则就变成了语义分析的一部分。

超过要分析程序检查的范围，但仍能被编译器检查的语言规则体称作语言的静态语义（static semantics）。它们包括类型检查（在一个静态分类的语言中）和诸如使用前先说明的规则。因此，只有当BNF规则可表达这些规则时才将其当作语法，而把其他的都当作语义。

除上下文有关文法之外，还有一种更常见的文法。这些文法称作非限制的文法（unrestricted grammar），它们具有格式 $A \rightarrow \alpha$ 的文法规则，在其中对格式 A 和 α 没有限制（除了不能是 ϵ 之外）。4种文法——非限制的、上下文有关的、上下文无关的和正则的，分别被称为0型、1型、2型和3型文法。它们构成的语言类称为以乔姆斯基命名的乔姆斯基层次（Chomsky hierarchy），这是因为乔姆斯基最先将它们用来描述自然语言。这些文法表示计算

能力的不同层次。非限制的（或 0 型）文法与图灵机完全相等，这与正则文法和有穷自动机相等是一样的，而且由此表示了绝大多数已知的计算。上下文无关文法也有一个对应的相等机器，即：下推自动机，但是对于分析算法而言，并不需要这种机器的全部能力，所以也就不再讨论它了。

我们还应留意一些上下文无关语言和文法相关的难处理的计算问题。例如，在处理二义性文法时，如能找到一个可将二义性文法转变成非二义性文法的算法并且又不会改变语言本身，那就太好了。然而不幸的是，大家都认为这是一个无法决定的事情，因此这样的算法是不可能存在的。实际上，甚至还存在着没有非二义性文法的上下文无关语言（称之为先天二义性语言（*inherently ambiguous language*）），而且判断一种语言是否是先天二义性的也是无法做到的。

幸运的是，程序设计语言并不会引起诸如先天二义性的复杂问题，而且也证明了经常谈到的消除二义性的特殊技术在应用中很合适。

3.7 TINY 语言的语法

3.7.1 TINY 的上下文无关文法

程序清单 3-1 是 TINY 在 BNF 中的文法，我们可从中观察到一些内容：TINY 程序只是一个语句序列，它共有 5 种语句：if 语句、repeat 语句、read 语句、write 语句和 assignment 语句。除了 if 语句使用 **end** 作为括号关键字（因此在 TINY 中没有悬挂 else 二义性）以及 if 语句和 repeat 语句允许语句序列作为主体之外，它们都具有类似于 Pascal 的语法，所以也就不需要括号或 **begin-end** 对（而且 **begin** 甚至在 TINY 中也不是一个保留字）。输入/输出语句由保留字 **read** 和 **write** 开始。read 语句一次只读出一个变量，而 write 语句一次只写出一个表达式。

程序清单 3-1 BNF 中的 TINY 的文法

```

program → stmt-sequence
stmt-sequence → stmt-sequence ; statement | statement
statement → if-stmt | repeat-stmt | assign-stmt | read-stmt | write-stmt
if-stmt → if exp then stmt-sequence end
           | if exp then stmt-sequence else stmt-sequence end
repeat-stmt → repeat stmt-sequence until exp
assign-stmt → identifier := exp
read-stmt → read identifier
write-stmt → write exp
exp → simple-exp comparison-op simple-exp | simple-exp
comparison-op → < | =
simple-exp → simple-exp addop term | term
addop → + | -
term → term mulop factor | factor
mulop → * | /
factor → ( exp ) | number | identifier

```

TINY 表达式有两类：在 if 语句和 repeat 语句的测试中使用比较算符 = 和 < 的布尔表达式，以及包括标准整型算符 +、-、* 和 /（它代表整型除法，有时也写作 div）的算术表达式（由文法中的 *simple-exp* 指出）。算术运算是左结合并有通常的优先关系。相反地，比较运算却是非结合的：每个没有括号的表达式只允许一种比较运算。比较运算比其他算术运算的优先权都低。

TINY中的标识符指的是简单整型变量，它没有诸如数组或记录构成的变量。TINY中也没有变量声明：它只是通过出现在赋值语句左边来隐式地声明一个变量。另外，它只有一个（全局）作用域，且没有过程或函数（因此也就没有调用）。

还要注意TINY的最后一个方面。语句序列必须包括将语句分隔开来的分号，且不能将分号放在语句序列的最后一个语句之后。这是因为TINY没有空语句（不同于Pascal和C）。另外，我们将stmt-sequence的BNF规则也写作一个左递归规则，但却并不真正在意语句序列的结合性，这是因为意图很简单，只需按顺序执行就行了。因此，只要将stmt-sequence编写成右递归即可。这个观点也出现在TINY程序的语法树结构中，其中的语法序列不是由树而是由列表表示的。现在就转到这个结构的讨论上来。

3.7.2 TINY编译器的语法树结构

TINY有两种基本的结构类型：语句和表达式。语句共有5类（if语句、repeat语句、assign语句、read语句和write语句），表达式共有3类（算符表达式、常量表达式和标识符表达式）。因此，语法树节点首先按照它是语句还是表达式来分类，接着根据语句或表达式的种类进行再次分类。树节点最大可有3个孩子的结构（仅在带有else部分的if语句中才需要它们）。语句通过同属域而不是使用子域来排序。

必须将树节点中的属性保留如下（除了前面所提到过的域之外）：每一种表达式节点都需要一个特殊的属性。常数节点需要它所代表的整型常数的域；标识符节点应包括了标识符名称的域；而算符节点则需要包括了算符名称的域。语句节点通常不需要属性（除了它们的节点类型之外）。但为了简便起见，在assign语句和read语句中，却要保留在语句节点本身中（除了作为一个表达式子节点之外）被赋予或被读取的变量名。

前面所描述的三个节点结构可通过程序清单3-2中的C说明得到，该说明还可在附录B（第198行到第217行）的globals.h文件的列表中找到。请注意我们综合了这些说明来帮助节省空间。它们还可帮助提醒每个节点类型的属性。现在谈谈说明中两个未曾提到过的属性。第1个是簿记属性lineno；它允许在转换的以后步骤中出现错误时能够打印源代码行数。第2个是type域，在后面的表达式（且仅是表达式）类型检查中会用到它。它被说明为枚举类型ExpType，第6章将会完整地讨论到它。

程序清单3-2 一个TINY语法树节点的C声明

```
typedef enum {StmtK, ExpK} NodeKind;
typedef enum {IfK, RepeatK, AssignK, ReadK, WriteK}
    StmtKind;
typedef enum {OpK, ConstK, IdK} ExpKind;

/* ExpType is used for type checking */
typedef enum {Void, Integer, Boolean} ExpType;

#define MAXCHILDREN 3

typedef struct treeNode
{
    struct treeNode * child[MAXCHILDREN];
    struct treeNode * sibling;
    int lineno;
    NodeKind nodekind;
    union { StmtKind stmt; ExpKind exp; } kind;
}
```

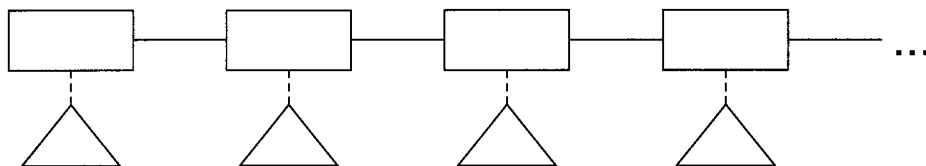


```

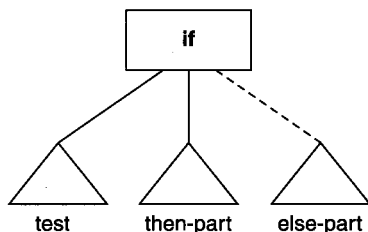
union { TokenType op;
        int val;
        char * name; } attr;
ExprType type; /* for type checking of exprs */
} TreeNode;

```

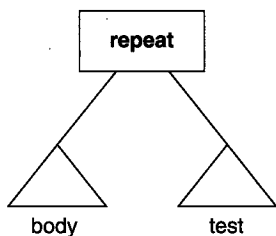
现在需要将语法树结构的描述用图形表示出来，并且画出示例程序的语法树。为了做到这一点，我们使用矩形框表示语句节点，用圆形框或椭圆形框表示表达式节点。语句或表达式的类型用框中的标记表示，额外的属性在括号中也列出来了。属性指针画在节点框的右边，而子指针则画在框的下面。我们还在图中用三角形表示额外的非指定的树结构，其中用点线表示可能出现也可能不出现的结构。语句序列由同属域连接（潜在的子树由点线和三角形表示）。则该图如下：



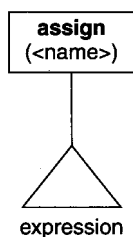
if 语句（带有3个可能的孩子）如下所示：



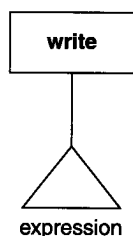
repeat 语句有两个孩子。第1个是表示循环体的语句序列，第2个是一个测试表达式：



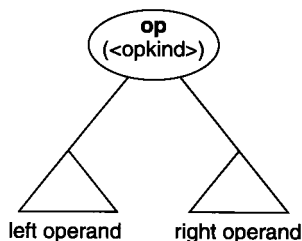
assign 语句有一个表示其值是被赋予的表达式的孩子（被赋予的变量名保存在语句节点中）：



write 语句也有一个孩子，它表示要写出值的表达式：



算符表达式有两个孩子，它们表示左操作数表达式和右操作数表达式：



其他所有的节点（read语句、标识符表达式和常量表达式）都是叶子节点。

最后准备显示一个 TINY 程序的树。程序清单 3-3 中有来自第 1 章计算一个整型阶乘的示例程序，它的语法树在图 3-6 中。

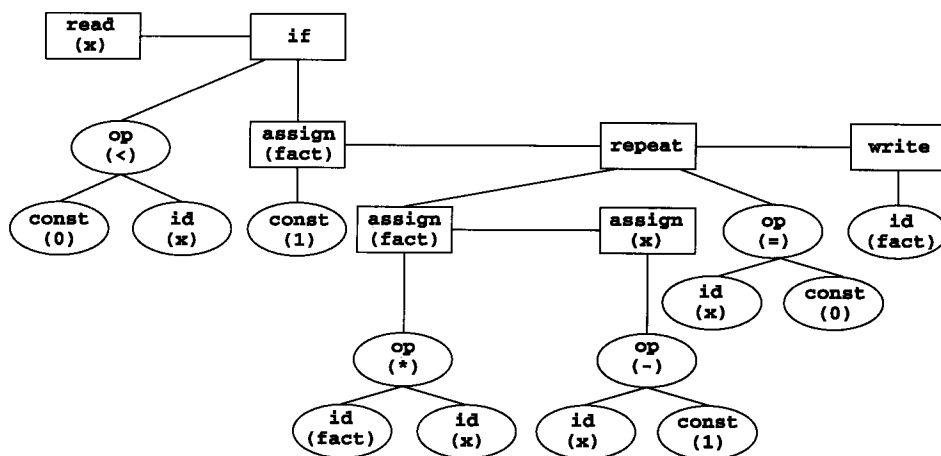


图3-6 程序清单3-3中TINY程序的语法树

程序清单 3-3 TINY语言中的示例程序

```

{ Sample program
  in TINY language-
  computes factorial
}
read x; { input an integer }
if 0 < x then { don't compute if x <= 0 }
  fact := 1;
  repeat
    fact := fact * x;
    x := x - 1
  
```

```

until x = 0;
write fact { output factorial of x }
end

```

练习

3.1 a. 写出一个生成串的集合

$\{s;, s;s;, s;s;s;, \dots\}$

的非二义性的文法。

b. 利用你的文法为串 $s;s$ 给出一个最左推导和最右推导。

3.2 假设有文法 $A \rightarrow AA \mid (A)\epsilon$

a. 描述它生成的语言。

b. 说明它有二义性。

3.3 假设有文法

```

exp    exp addop term | term
addop  + | -
term    term mulop factor | factor
mulop  *
factor  ( exp ) | number

```

则为下面的表达式写出最左推导、分析树以及抽象语法树：

a. $3+4*5-6$

b. $3*(4-5+6)$

c. $3-(4+5*6)$

3.4 下面的文法生成字母表之上的所有正则表达式（以前曾在算符前后加上了引号，这是因为竖线既是一个算符又是一个元字符）：

```

rexp    rexp "|" rexp
        | rexp rexp
        | rexp "*"
        | "(" rexp ")"
        | letter

```

a. 利用这个文法为正则表达式 $(ab|b)^*$ 给出一个推导。

b. 说明该文法有二义性。

c. 重写该文法以使算符建立正确的优先关系（参见第2章）。

d. (c) 的答案给二进制算符带来怎样的结合性？为什么？

3.5 为包括了常量 **true** 和 **false**、算符 **and**、**or** 和 **not**，以及括号的布尔表达式编写一个文法。确保给予 **or** 比 **and** 低的优先权，而 **and** 的优先权比 **not** 低，并允许 **not** 重复使用，如在布尔表达式中的 **not not true**。另外还需保证该文法没有二义性。

3.6 考虑以下表示简化的类LISP表达式的文法：

```

lexp    atom | list
atom    number | identifier
list    ( lexp-seq )
lexp-seq lexp-seq lexp | lexp

```

a. 为串 $(a\ 2\ 3\ (m\ x\ y))$ 分别写出一个最左推导和一个最右推导。

b. 为a部分中的串画出一个分析树。

3.7 a. 为练习3.6中语法的抽象语法树结构编写一个C类型声明。

b. 为从a部分的声明得出的串(a 23 (m x y))画出一个语法树。

3.8 假设以下的文法

```
statement  if-stmt | other | ε
if-stmt   if ( exp ) statement else-part
else-part else statement | ε
exp       0 | 1
```

a. 为串

```
if(0) if(1)other else elseother
```

画出分析树。

b. 用两个else的目的是什么？

c. 在C中允许有这样的代码吗？请解释原因。

3.9 (Aho, Sethi and Ullman) 显示以下的解决悬挂 else 二义性问题的尝试仍存在着二义性 (与3.4.3节中的解法对比一下)：

```
statement  if ( exp ) statement | matched-stmt
matched-stmt  if ( exp ) matched-stmt else statement | other
exp         0 | 1
```

3.10 a. 将练习3.6中的文法翻译成EBNF。

b. 为a部分中的EBNF画出语法图。

3.11 假设集合等式 $X = (X + X) \ N$ (N是自然数集：参见3.6.2节)。

a. 说明集合 $E = N \ (N + N) \ (N + N + N) \ (N + N + N + N) \ \dots$ 满足等式。

b. 假设任意一个集合 E' 满足等式，说明有 $E \subseteq E'$ 成立。

3.12 可以用多种方法将一目减添加到练习3.3的简单算术表达式文法中。修改BNF以使其符合每一个要求的规则。

a. 每个表达式至多允许有一个一目减，且一目减应在表达式的开头，则 $-2-3$ 是合法的^① (且答案应为-5)， $-2-(-3)$ 也应是正规的，但 $-2--3$ 却是非法的。

b. 在一个数或左括号之前至多有一个一目减，所以 $-2--3$ 是合法的，但 $--2$ 和 $-2---3$ 却是非法的。

c. 在数字和左括号之前允许有任意数量的一目减，所以任何情况都是合法的。

3.13 考虑将练习3.6的文法如下简化：

```
lexp      number | ( op lexp-seq )
op        + | - | *
lexp-seq  lexp-seq lexp | lexp
```

这个文法可被认为是表示在类似于 LISP 的前缀格式中的简单整型算术表达式。

例如，表达式 $34-3*42$ 按该文法就可写成 $(-34 \ (*3 \ 42))$

a. 正规表达式 $(- \ 2 \ 3 \ 4)$ 与 $(- \ 2)$ 的解释是什么？表达式 $(+ \ 2)$ 与 $(* \ 2)$ 的解释是什么呢？

① 请注意这个表达式中的第2个减法是一个二目减，而不是一目减。

b. 在这个文法中，有没有优先权和结合性的问题？该文法有二义性吗？

3.14 a. 为上题中文法的语法树结构写出C类型的说明。

b. 利用a部分的答案为表达式 $(- 34 (* 3 42))$ 画出语法树。

3.15 在3.3.2节中，一个抽象语法树

$$\begin{aligned} \text{exp} & \quad \text{OpExp}(\text{op}, \text{exp}, \text{exp}) \mid \text{ConstExp}(\text{integer}) \\ \text{op} & \quad \text{Plus} \mid \text{Minus} \mid \text{Times} \end{aligned}$$

的类似于BNF的说明与一些语言中的真正类型说明很接近。ML和Haskell是其中的两种语言。这个练习是为那些了解这两种语言中任一种的读者编写的。

a. 写出完成上面抽象语法的数据类型说明。

b. 为表达式 $(34 * (42 - 3))$ 写出一个抽象语法表达式。

3.16 重写3.3.2节开始的语法树typedef以便使用一个union。

3.17 证明练习3.5中的文法生成了成对括号的所有串的集合，其中假设 w 具有以下两个属性，那么 w 就是一个成对括号的串：

a. w 确实包括了相同数量的左括号和右括号。

b. w 的每个前缀 u （对于某个 x ，有 $w = ux$ ）至少有与右括号相同的左括号（提示：通过归纳一个推导的长度来证明）。

3.18 a. 写出一个生成格式 xcx 的串的文法，其中 x 是 a 和 b 的一个串。

b. 有没有可能为a部分的串写出一个上下文无关的文法？为什么？

3.19 在一些语言（例如Modula-2和Ada）中，希望一个过程说明用包括了过程名的语法结束，在Modula-2中，一个过程是这样说明的：

```
PROCEDURE P;
BEGIN
...
END P;
```

请注意在END之后的过程名P。分析程序能够检查这样的要求吗？为什么？

3.20 a. 写出一个生成与下面文法相同的语言的正则表达式：

$$\begin{aligned} A & \quad aA \mid B \mid \varepsilon \\ B & \quad bB \mid A \end{aligned}$$

b. 写出一个生成与下面正则表达式相同的语言的文法：

$$(a \mid c \mid ba \mid bc)^*(b \mid \varepsilon)$$

3.21 单元产生式（unit production）是公式 $A \rightarrow B$ 的一个文法规则选择，其中 A 和 B 均为非终结符。

a. 说明可系统地删除单元产生式，这样就产生了一个不带有单元产生式的文法，该单元产生式就生成了与原始文法相同的语言。

b. 你希望单元产生式在程序设计语言中经常出现吗？为什么？

3.22 循环文法（cyclic grammar）是在其中有一些非终结符 A 的一个推导 $A \Rightarrow^* A$ 的文法。

a. 说明循环文法是有二义性的。

b. 你希望定义程序设计语言的文法经常是循环的吗？请解释原因。

3.23 将练习3.6的TINY文法重写为在EBNF中的文法。

3.24 对于TINY程序

```
read x;
```

```
x:=x+1;  
write x
```

- a. 画出TINY分析树。
- b. 画出TINY语法树。

3.25 为下面的程序画出TINY语法树：

```
read u;  
read v; { input two integers }  
if v=0 then v:=0 { do nothing }  
else  
  repeat  
    temp:=v;  
    v:=u-u/v*v;  
    u:=temp  
  until v=0  
end;  
write u {output gcd of original u & v}
```

注意与参考

上下文无关文法的许多理论都可在 Hopcroft and Ullman [1979] 中找到；在这里还可找到一些尚未解决的问题，例如先天二义性这样的许多属性；此外还有乔姆斯基层次。其他的内容则可在 Ginsburg [1966, 1975] 中找到。早期的一些理论在 Chomsky [1956, 1959] 中有提到，它为自然语言的学习提供了帮助。只有到后来对于程序语言的理解才成为了一个重要的论题，上下文无关文法的第1个应用是在 Algol60 的定义之中（Naur [1963]）。使用了上下文无关规则的有关 3.4.3 节中的悬挂 else 问题的解决方法是取自 Aho、Hopcroft 和 Ullman [1986] 的，练习 3.9 的答案也来自这儿。将上下文无关文法看作是递归等式（3.6.2 节）是由指示语义得到的，读者可参看 Schmidt [1986]。

第4章 自顶向下的分析

本章要点

- 使用递归下降分析算法进行自顶向下的分析
- LL(1)分析
- First 集合和Follow集合
- TINY 语言的递归下降分析程序
- 自顶向下分析程序中的错误校正

自顶向下 (top-down) 的分析算法通过在最左推导中描述出各个步骤来分析记号串输入。之所以称这样的算法为自顶向下是由于分析树隐含的编号是一个前序编号, 而且其顺序是由根到叶 (参见第3章的3.3节)。自顶向下的分析程序有两类: 回溯分析程序 (backtracking parser) 和预测分析程序 (predictive parser)。预测分析程序试图利用一个或多个先行记号来预测出输入串中的下一个构造, 而回溯分析程序则试着分析其他可能的输入, 当一种可能失败时就要求输入中备份任意数量的字符。虽然回溯分析程序比预测分析程序强大许多, 但它们都非常慢, 一般都在指数的数量级上, 所以对于实际的编译器并不合适。本书将不研究回溯程序 (但读者可查看“注意与参考”部分以及练习以得到一些关于这个主题的提示)。

本章要学习的两类自顶向下分析算法分别是递归下降分析 (recursive-descent parsing) 和LL(1)分析 (LL(1) parsing)。递归下降分析很常用, 且它对于手写的分析程序最为适合, 所以我们最先学习它。之后再学习 LL(1)分析, 由于在实际中并不常用到它, 所以只是将其作为一个带有显式栈的简单实例来学习, 它是下一章更强大 (但也更复杂) 的自底向上算法的前奏。它对于将出现在递归下降分析中的一些问题形式化也有帮助。LL(1)分析方法是这样得名的: 第1个“L”指的是由左向右地处理输入 (一些旧式的分析程序惯于自右向左地处理输入, 但现在已不常用了)。第2个“L”指的是它为输入串描绘出一个最左推导。括号中的数字1意味着它仅使用输入中的一个符号来预测分析的方向 (“LL(k)分析”也是有可能的——它利用向前看的 k 个符号, 本章后面将简要地介绍到它, 但是向前看的一个符号是最为常见的)。

递归下降程序分析和 LL(1)分析一般地都要求计算先行集合, 它们分别称作 First集合和Follow集合[⊖]。由于无需显式地构造出这些集合就可以构造出简单的自顶向下的分析程序, 所以在基本算法的介绍之后我们再讨论它们。之后我们还要谈到一个由递归下降分析构造的 TINY 分析程序, 本章的最后是自顶向下的分析中的错误校正。

4.1 使用递归下降分析算法进行自顶向下的分析

4.1.1 递归下降分析的基本方法

递归下降分析的概念极为简单: 将一个非终结符 A 的文法规则看作将识别 A 的一个过程的定义。A 的文法规则的右边指出这个过程的代码结构: 一个选择中的终结符与非终结符序列与

[⊖] 下一章将要研究的自底向上的分析算法有一些也需要这些集合。

相匹配的输入以及对其他过程的调用相对应，而选择与在代码中的替代情况（`case`语句和`if`语句）相对应。

例如，考虑前一章的表达式文法：

```
exp    exp addop term | term
addop  + | -
term   term mulop factor | factor
mulop  *
factor ( exp ) | number
```

及`factor`的文法规则，识别`factor`并用相同名称进行调用的递归下降程序过程可用伪代码编写如下：

```
procedure factor ;
begin
  case token of
    ( : match( ) ;
      exp ;
      match( ) ;
    number :
      match (number) ;
    else error ;
  end case ;
end factor ;
```

在这段伪代码中，假设有一个在输入中保存当前下一个记号的`token`变量（以便这个例子使用先行的一个符号）。另外还假设有一个`match`过程，它用它的参数匹配当前的下一个记号。如果成功则前移，如果失败就表明错误：

```
procedure match ( expectedToken ) ;
begin
  if token = expectedToken then
    getToken ;
  else
    error ;
  end if ;
end match ;
```

现在脱离开在`match`和`factor`中被调用的未指定的`error`过程。可以假设它会打印出一个出错信息并退出。

请注意，在`match ()`调用和`factor`中的`match (number)`调用中，我们知道`expectedToken`和`token`是一样的。但是在`match ()`调用中，不能将`token`假设为一个右括号，所以需要有一个测试。`factor`的代码也假设已将过程`exp`定义为可以调用。在表达式文法的递归下降分析中，`exp`过程将调用`term`，`term`过程将调用`factor`，而`factor`过程将调用`exp`，所以所有的这些过程都必须能够互相调用。不幸的是，为表达式文法中的其余规则编写递归下降程序过程并不像为`factor`编写一样简单，而且它需要使用EBNF，下面就介绍这一点。

4.1.2 重复和选择：使用EBNF

例如，一个if语句（简化了的）文法规则是：

$$\begin{aligned} \text{if-stmt} \quad & \mathbf{if} \ (\ exp \) \ statement \\ & | \ \mathbf{if} \ (\ exp \) \ statement \ \mathbf{else} \ statement \end{aligned}$$

可将它翻译成以下过程

```

procedure ifStmt ;
begin
    match ( if ) ;
    match ( ( ) ) ;
    exp ;
    match ( ) ) ;
    statement ;
    if token = else then
        match ( else ) ;
        statement ;
    end if ;
end ifStmt ;

```

在这个例子中，不能立即区分出文法规则右边的两个选择（它们都以记号 **if** 开始）。相反地，我们必须直到看到输入中的记号 **else** 时，才能决定是否识别可选的 **else** 部分。因此，if语句的代码与EBNF

$$\text{if-stmt} \quad \mathbf{if} \ (\ exp \) \ statement \ [\ \mathbf{else} \ statement \]$$

匹配的程度比与BNF的匹配程序要高，上面的EBNF的方括号被翻译成ifStmt的代码中的一个测试。实际上，EBNF表示法是为更紧密地映射递归下降分析程序的真实代码而设计的，如果使用的是递归下降程序，就应总是将文法翻译成EBNF。另外还需注意到即使这个文法有二义性（参见前一章），编写一个每当在输入中遇到 **else** 记号时就立即匹配它的分析程序也是很自然的。这与最近嵌套的消除二义性的规则精确对应。

现在考虑一下BNF中简单算术表达式文法中的 *exp* 情况：

$$\text{exp} \quad \text{exp addop term} \mid \text{term}$$

如要根据我们的计划试着将它变成一个递归的 *exp* 过程，则首先应做的是调用 *exp* 本身，而这将立即导致一个无限递归循环。由于 *exp* 和 *term* 可以以相同的记号开头（一个数或左括号），所以要想测试使用哪个选择（*exp* *exp addop term* 或 *exp* *term*）就会出现问题了。

解决的办法是使用EBNF规则

$$\text{exp} \quad \text{term} \{ \text{addop term} \}$$

花括号表示可将重复部分翻译到一个循环的代码中，如下所示：

```

procedure exp ;
begin
    term ;
    while token = + or token = - do

```

```

    match (token);
    term ;
end while ;
end exp ;

```

相似地，*term*的EBNF规则：

$$term \rightarrow factor \{ mulop factor \}$$

就变成代码

```

procedure term ;
begin
    factor ;
    while token = * do
        match (token);
        factor ;
    end while ;
end term ;

```

在这里当分隔过程时，删去了非终结符 *addop* 和 *mulop*，这是因为它们仅有匹配算符功能：

$$\begin{array}{ll} addop & + \mid - \\ mulop & * \end{array}$$

这里所做的是 *exp* 和 *term* 中的匹配。

这个代码有一个问题：由花括号（和原始的BNF中的显式）表示的左结合是否仍然保留。例如，假设要为本书中简单整型算术的文法编写一个递归下降程序计算器，就可通过在循环中轮转来完成运算，从而就保证了该运算是左结合（现在假设分析过程是返回一个整型结果的函数）：

```

function exp : integer ;
var temp : integer ;
begin
    temp := term ;
    while token = + or token = - do
        case token of
            + : match (+);
                temp := temp + term ;
            - : match (-);
                temp := temp - term ;
        end case ;
    end while ;
    return temp ;
end exp ;

```

term 与之也类似。我们已利用这种方法创建了一个可运行的简单计算器，程序清单 4-1 中有它

的C代码。其中并未写出一个完整的扫描程序，而是选择使用了对 `getchar` 和 `scanf` 的调用来代替 `getToken` 的过程。

程序清单4-1 简单整型算术的递归下降程序计算器

```

/* Simple integer arithmetic calculator
   according to the EBNF:

   <exp> -> <term> { <addop> <term> }
   <addop> -> + | -
   <term> -> <factor> { <mulop> <factor> }
   <mulop> -> *
   <factor> -> ( <exp> ) | Number

   Inputs a line of text from stdin
   Outputs "Error" or the result.
*/

#include <stdio.h>
#include <stdlib.h>

char token; /* global token variable */

/* function prototypes for recursive calls */
int exp(void);
int term(void);
int factor(void);

void error(void)
{ fprintf(stderr, "Error\n");
  exit(1);
}

void match( char expectedToken)
{ if (token==expectedToken) token = getchar();
  else error();
}

main()
{ int result;
  token = getchar(); /* load token with first
                     character for lookahead */

  result = exp();
  if (token=='\n') /* check for end of line */
    printf("Result = %d\n", result);
  else error(); /* extraneous chars on line */
  return 0;
}

int exp(void)
{ int temp = term();
  while ((token=='+') || (token=='-'))
    switch (token) {
      case '+': match('+');
                temp+=term();
                break;

```

```

        case '-': match('-');
                temp-=term();
                break;
    }
    return temp;
}

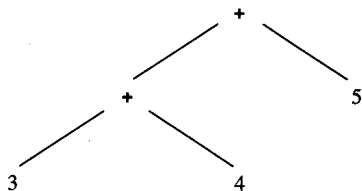
int term(void)
{ int temp = factor();
  while (token=='*') {
      match('*');
      temp*=factor();
  }
  return temp;
}

int factor(void)
{ int temp;
  if (token=='(') {
      match('(');
      temp = exp();
      match(')');
  }
  else if (isdigit(token)) {
      ungetc(token,stdin);
      scanf("%d",&temp);
      token = getchar();
  }
  else error();
  return temp;
}

```

这个将EBNF中的文法规则转变成代码的办法十分有效，4.4节还将利用它为TINY语言给出一个完整的分析程序。但是它仍有一些缺点，且须留意在代码中安排的动作。例如：在前面`exp`的伪代码中，运算的匹配必须发生在对`term`的重复调用之前（否则`term`会将一个运算看作是它的第1个记号，这就会生成一个错误了）。实际上现在必须严格遵循用于保持`token`变量的协议：必须在分析开始之前先设置`token`，且对一个记号的测试（将它放在伪代码的`match`过程中）一旦成功，就立即调用`getToken`（或它的等价物）。

在构造语法树中也须注意动作的安排。我们已看到可通过在执行循环时完成计算来保持带有重复的EBNF的左结合。它再也不能与分析树或语法树中的自顶向下的构造相对应。但相反地，如考虑表达式`3+4+5`，其语法树



在根节点（表示其与5的和的节点）之前应先建立表示3与4之和的节点。将它翻译为真正的语法树结构就有了以下的`exp`过程的伪代码：

```

function exp : syntaxTree ;
var temp, newtemp : syntaxTree ;
begin
    temp := term ;
    while token = + or token = - do
        case token of
            + : match (+) ;
                newtemp := makeOpNode(+) ;
                leftChild(newtemp) := temp ;
                rightChild(newtemp) := term ;
                temp := newtemp ;
            - : match (-) ;
                newtemp := makeOpNode(-) ;
                leftChild(newtemp) := temp ;
                rightChild(newtemp) := term ;
                temp := newtemp ;
        end case ;
    end while ;
    return temp ;
end exp ;

```

或更简单的

```

function exp : syntaxTree ;
var temp, newtemp : syntaxTree ;
begin
    temp := term ;
    while token = + or token = - do
        newtemp := makeOpNode(token) ;
        match (token) ;
        leftChild(newtemp) := temp ;
        rightChild(newtemp) := term ;
        temp := newtemp ;
    end while ;
    return temp ;
end exp ;

```

这个代码使用了新函数 *makeOpNode*，它接受作为参数的运算符记号并返回新建的语法树节点。我们还通过写出 *leftChild* (*t*) := *p* 或 *rightChild* (*t*) := *p* 指出将一个语法树 *p* 的赋值作为语法树 *t* 的一个左子树或右子树。有了这个伪代码之后，*exp* 过程实际构造了语法树而不是分析树。这是因为一个对 *exp* 的调用并不总是构造一个新的树节点；如没有运算符，*exp* 则仅仅是传送回从最初调用到 *term* 收到的树来作为它自己的值。也可以写出与 *term* 和 *factor* 相对应的伪代码（参见练习）。

相反地，递归下降分析程序在严格的自顶向下的风格中可构造出 if 语句的语法树：

```
function ifStatement : syntaxTree ;
var temp : syntaxTree ;
begin
    match (if) ;
    match ( ( ) ;
    temp := makeStmtNode(if) ;
    testChild(temp) := exp ;
    match ( ) ;
    thenChild(temp) := statement ;
    if token = else then
        match (else) ;
        elseChild(temp) := statement ;
    else
        elseChild(temp) := nil ;
    end if ;
end ifStatement ;
```

正因为递归下降分析允许程序设计人员可调整动作的安排，这样它就可以选择手工生成的分析程序了。

4.1.3 其他决定问题

前面描述的递归下降分析虽然非常强大，但它仍有特殊性。若使用的是一个设计精巧的小型语言（例如 TINY，甚至是 C），那么对于构造一个完整的分析程序，这些办法是适合的。读者应注意在复杂情况中还需要更形式的方法。此时还会出现一些问题。首先，将原先在 BNF 中编写的文法转变成 EBNF 格式可能会有些困难。下一节将学习不用 EBNF 的另一种方法，其中构造了一个与 EBNF 基本相等的转变了的 BNF。其次，在用公式表达一个用以区分两个或更多的文法规则选项的测试

$$A \quad | \quad | \dots$$

时，如果 和 均以非终结符开始，那么就很难决定何时使用 A 选项，何时又使用 A 选项。这个问题就要求计算 和 的 First 集合：可以正规地开始每个串的记号集合。4.3 节将详细介绍这个计算。再次，在写 ϵ 产生式的代码

$$A \quad \epsilon$$

时，需要了解什么记号可以正规地出现在非终结符 A 之后，这是因为这样的记号指出 A 可以恰当地在分析中的这个点处消失。这个集合被称作 A 的 Follow 集合。4.3 节中也有这个集合的准确计算。

最后读者需要注意：人们进行 First 集合和 Follow 集合的计算是为了对早期错误进行探测。例如：在程序清单 4-1 的计算器程序中，假设有输入) 3 - 2)，分析程序在报告错误之前，将从 exp 到 term 再到 factor 下降；由于在表达式中，右括号作为第 1 个字符并不合法，而在 exp 中可能早已声明了这个错误。exp 的 First 集合将会告诉我们这一点，从而可以进行较早的错误探测（本章最后将更详细地讨论错误探测和恢复）。

4.2 LL(1)分析

4.2.1 LL(1)分析的基本方法

LL(1)分析使用显式栈而不是递归调用来完成分析。以标准方式表示这个栈非常有用，这样LL(1)分析程序的动作就可以快捷地显现出来。在这个介绍性的讨论中，我们使用了生成成对括号的串的简单文法：

$$S \rightarrow (S) S \mid \epsilon$$

(参见第3章的例3.5)。

假设有这个文法和串 $(())$ ，则表4-1给出了自顶向下的分析程序的动作。此表共有4列。第1列为便于以后参考给每个步骤标上了号码。第2列显示了分析栈的内容，栈的底部向左，栈的顶部向右。栈的底部标注了一个美元符号，这样一个在顶部包含了非终结符 S 的栈就是：

$$\$ S$$

且将额外的栈项推向右边。表的第3列显示了输入。输入符号由左列向右。美元符号标出了输入的结束（它与由扫描程序生成的EOF记号相对应）。表的第4列给出了由分析程序执行的动作的简短描述，它将改变栈和（有可能）输入，如在表的下一行中所示一样。

表4-1 自顶向下的分析程序的分析动作

分析栈	输入	动作
1 \$ S	() \$	$S \rightarrow (S) S$
2 \$ S) S (() \$	匹配
3 \$ S) S) \$	$S \rightarrow \epsilon$
4 \$ S)) \$	匹配
5 \$ S	\$	$S \rightarrow \epsilon$
6 \$	\$	接受

自顶向下的分析程序是从将开始符号放在栈中开始的。在一系列动作之后，它接受一个输入串，此时栈和输入都空了。因此，成功的自顶向下的分析的一般示意法应是：

```
$ StartSymbol      InputString $
...
...
$                  $ accept
```

在上面的例子中，开始符号是 S ，输入串是 $(())$ 。

自顶向下的分析程序通过将栈顶部的非终结符替换成文法规则中（BNF中）该非终结符的一个选择来作出分析。其方法是在分析栈的顶部生成当前输入记号，在顶部它已匹配了输入记号并将它从栈和输入中舍弃掉。这两个动作

- 1) 利用文法选择 A 将栈顶部的非终结符 A 替换成串。
- 2) 将栈顶部的记号与下一个输入记号匹配。

是自顶向下的分析程序中的两个基本动作。第1个动作称为生成（generate）：通过写出在替换中使用的BNF选择（它的左边在当前必须是栈顶部的非终结符）来指出这个动作。第2个动作将栈顶部的一个记号与输入中的下一个记号匹配（并通过取出栈和将输入向前推进而将二者全

部舍弃掉)；这个动作是通过书写单词来指出的。另外还需注意在生成动作中，从BNF中替换掉的串必须颠倒地压在栈中（这是因为要保证串按自左向右的顺序进到栈的顶部）。

例如，在表4-1的分析的第1步中，栈和输入分别是

$$\$ S \quad () \$$$

且用来替换栈顶部的S的规则是 $S \rightarrow (S) S$ ，所以将串 $S) S ($ 压入到栈中得到

$$\$ S) S (() \$$$

现在已生成了下一个输入终结符，即在栈的顶部的一个左括号，我们还完成了一个匹配以得到以下的情况：

$$\$ S) S) \$$$

表4-1中生成动作的列表与串 $()$ 最左推导的步骤完全对应：

$S \rightarrow (S) S$	$[S \rightarrow (S) S]$
$() S$	$[S \rightarrow \epsilon]$
$()$	$[S \rightarrow \epsilon]$

这是自顶向下分析的特征。如果要在分析进行时构造一个分析树，则可当将每个非终结符或终结符压入到栈中时添加节点来构造动作。因此分析树根节点的构造是在分析开始时进行的（与开始符号对应）。而在表4-1的第2步中，当 $(S) S$ 替换S时，用于4个替换符号中的每个符号的节点将作为放到栈中的符号来构造，并且作为子节点与在栈中替换的S节点连接。为了使其具有高效率，就必须修改栈以包括指向这些构造的节点的指针，而不是仅仅是指向非终结符或终结符本身的指针。另外，读者还将看到如何将这个处理进行修改以生成语法树的结构而非分析树的结构。

4.2.2 LL(1)分析与算法

当非终结符A位于分析栈的顶部时，根据当前的输入记号（先行），必须使用刚刚描述过的分析办法做出一个决定：当替换栈中的A时应为A选择哪一个文法规则，相反地，当记号位于栈顶部时，就无需做出这样的决定，这是因为无论它是当前的输入记号（由此就发生一个匹配），还是不是输入记号（从而就发生一个错误），两者都是相同的。

通过构造一个LL(1)分析表(LL(1) parsing table)就可以表达出可能的选择。这样的表格基本上是一个由非终结符和终结符索引的二维数组，其中非终结符和终结符包括了要在恰当的分析步骤（包括代表输入结束的 $\$$ ）中使用的产生式选择。这个表被称为 $M[N, T]$ ，这里的N是文法的非终结符的集合，T是终结符或记号的集合（为了简便，禁止将 $\$$ 加到T上），M可被认为是“运动的”表。我们假设表 $M[N, T]$ 在开始时，它的所有项目均为空。任何在构造之后继续为空的项目都代表了在分析中可能发生的潜在错误。

根据以下规则在这个表中添加产生式：

- 1) 如果A是一个产生式选择，且有推导 $A \Rightarrow^* a$ 成立，其中a是一个记号，则将A添加到表项目 $M[A, a]$ 中。
- 2) 如果A是一个产生式选择，且有推导 $A \Rightarrow^* \epsilon$ 和 $S \Rightarrow^* Aa$ 成立，其中S是开始符号，a是一个记号（或 $\$$ ），则将A添加到表项目 $M[A, a]$ 中。

这个规则的观点是：在规则1中，在输入中给出了记号a，若可为匹配生成一个a，则希望挑选规则A。在规则2中，若A派生了空串（通过A），且如a是一个在推导中可合法地出

现在 A 之后的记号,则要挑选 A 以使 A 消失。请注意规则2中当 $=\varepsilon$ 时的特殊情况。

很难直接完成这些规则。在下一节中,我们将为此开发一个算法,这其中包括了早已提到过的First和Follow集合。但在极为简单的例子中,这些规则是可手工完成的。

读者可考虑在前一小节中使用成对括号的文法的第1个示例,它有一个非终结符(S)、3个记号(左括号、右括号和 $\$$),以及两个选择。由于 S 只有一个非空产生式,即 $S \rightarrow (S)S$,每一个可从 S 派生的串必须或为空或以左括号开始,并将这个产生式选择添加到项目 $M[S, (]$ 中(且仅能在这里)。这样就完成了规则1之下的所有情况。因为有 $S \rightarrow (S)S$,规则2应用了 $=\varepsilon$, $= (, A = S, a =)$ 且 $= S \$$,所以 $S \rightarrow \varepsilon$ 就被添加到 $M[S,)]$ 中了。由于 $S \$ \rightarrow *S\$$ (空推导),所以 $S \rightarrow \varepsilon$ 也被添加到 $M[S, \$]$ 中。这样就完成了这个文法的LL(1)分析表的构造,我们可以将它写在下面的格式中:

$M[N, T]$	()	\$
S	$S \rightarrow (S)S$	$S \rightarrow \varepsilon$	$S \rightarrow \varepsilon$

为了完善LL(1)分析算法,该表必须为每个非终结符-记号对给出唯一选择,可以从下面的定义开始。

定义:如果文法 G 相关的LL(1)分析表的每个项目中至多只有一个产生式,则该文法就是LL(1)文法(LL(1) grammar)。

由于上面的定义暗示着利用LL(1)文法表就能构造出一个无二义性的分析,所以LL(1)文法不能是二义性的。当给出LL(1)文法时,程序清单4-2中有使用LL(1)分析表的一个分析算法。这个算法完全导致了在前一小节的示例中描述的那些动作。

虽然程序清单4-2中的算法要求分析表的每个项目最多只能有一个产生式,但仍有可能在表构造中建立消除二义性的规则,它可处理简单的二义性情况,如在与递归下降程序相似的方式中的悬挂else问题。

程序清单4-2 基于表的LL(1)分析算法

```
(* assumes $ marks the bottom of the stack and the end of the input *)
push the start symbol onto the top of the parsing stack ;
while the top of the parsing stack  $\neq \$$  and the next input token  $\neq \$$  do
    if the top of the parsing stack is terminal  $a$ 
        and the next input token =  $a$ 
    then (* match *)
        pop the parsing stack ;
        advance the input ;
    else if the top of the parsing is nonterminal  $A$ 
        and the next input token is terminal  $a$ 
        and parsing table entry  $M[A, a]$  contains
            production  $A \rightarrow X_1X_2 \dots X_n$ 
    then (* generate *)
        pop the parsing stack ;
        for  $i := n$  downto 1 do
            push  $X_i$  onto the parsing stack ;
        else error ;
if the top of the parsing stack =  $\$$ 
    and the next input token =  $\$$ 
then accept
else error ;
```

例如，if语句简化了的文法（参见第3章3.6节）：

```

statement  if-stmt | other
if-stmt   if ( exp ) statement else-part
else-part  else statement | ε
exp       0 | 1

```

构造LL(1)分析表就得出了表4-2中的结果，我们并未在表中列出括号终结符（或），这是因为它们并不引起动作（将在下一节中详细地解释这个表的构造）。

表4-2 （二义性的）if语句的LL(1)分析表

$M[N, T]$	if	other	else	0	1	\$
statement	statement if-stmt	statement other				
if-stmt	if-stmt if (exp) statement else-part					
else-part			else-part else statement else-part ε			else-part ε
exp				Exp 0	exp 1	

在表4-2中，项目 $M[\text{else-part}, \text{else}]$ 包括了两个项目，且它与悬挂 else 二义性对应。与在递归下降程序中一样，当构造这个表时，我们可以提供总是倾向于生成当前先行记号规则的消除二义性的规则，则倾向于产生式

else-part **else** statement

而不是产生式 *else-part* ε。这实际上与最接近嵌套消除二义性规则对应。通过这个修改，表4-2就变成无二义性的了，而且可以对文法进行分析，这就好像它是一个 LL(1)文法一样。例如，表4-3显示了LL(1)分析算法的分析动作，它给出了串

if(0) if(1) other else other

（为了简便，我们将图中的词进行缩写：*statement* = *S*、*if-stmt* = *I*、*else-part* = *L*、*exp* = *E*、**if** = **i**、**else** = **e**、**other** = **o**）。

表4-3 为if语句使用最接近嵌套消除二义性规则的LL(1)分析

分 析 栈	输 入	动 作
\$ S	i (0) i (1) o e o \$	S I
\$ I	i (0) i (1) o e o \$	I i (E) S L
\$ L S) E (i	i (0) i (1) o e o \$	匹配
\$ L S) E ((0) i (1) o e o \$	匹配
\$ L S) E	0) i (1) o e o \$	E 0
\$ L S) o	0) i (1) o e o \$	匹配

(续)

分析栈	输入	动作
\$LS)) i (1) o e o \$	匹配
\$LS	i (1) o e o \$	$S \rightarrow I$
\$LI	i (1) o e o \$	$I \rightarrow i (E) SL$
\$LLS) E (i	i (1) o e o \$	$I \rightarrow i (E) SL$
\$LLS) E (i	i (1) o e o \$	匹配
\$LLS) E ((1) o e o \$	匹配
\$LLS) E	1) o e o \$	$E \rightarrow 1$
\$LLS)) o e o \$	匹配
\$LLS	o e o \$	$S \rightarrow o$
\$LL o	o e o \$	匹配
\$LL	e o \$	$L \rightarrow e S$
\$LS e	e o \$	匹配
\$LS	o \$	$S \rightarrow o$
\$L o	o \$	匹配
\$L	\$	$L \rightarrow \epsilon$
\$	\$	接受

4.2.3 消除左递归和提取左因子

LL(1)分析中的重复和选择也存在着与在递归下降程序分析中遇到的类似问题，而且正是由于这个原因，还不能够为前一节的简单算法表达式文法给出一个 LL(1)分析表。我们利用 EBNF 表示法解决了递归下降程序中的这些问题，但却不能在 LL(1)分析中使用相同的办法；而必须将 BNF 表示法中的文法重写到 LL(1)分析算法所能接受的格式上。此时应用的两个标准技术是左递归消除 (left recursion removal) 和提取左因子 (left factoring)。我们将逐个考虑它们。必须指出的是：这两个技术无法保证可将一个文法变成 LL(1)文法，这就同 EBNF 一样无法保证在编写递归下降程序中可以解决所有的问题。然而，在绝大多数情况下，它们都十分有用，并且具有可自动操作其应用程序的优点，因此，假设有一个成功的结果，利用它们就可自动地生成 LL(1)分析程序 (参见“注意与参考”一节)。

1) 左递归消除 左递归被普遍地用来运算左结合，如在简单表达式文法中，

$$exp \rightarrow exp \text{ addop } term \mid term$$

使得运算由 *addop* 左结合来代表。这是左递归的最简单情况，在其中只有一个左递归产生式选择。当有不止一个的选择是左递归时，就略微复杂一些了，如可将 *addop* 写出来：

$$exp \rightarrow exp + term \mid exp - term \mid term$$

这两种情况都涉及到了直接左递归 (immediate left recursion)，而左递归仅发生在一个非终结符 (如 *exp*) 的产生式中。间接左递归是更复杂的情况，如在规则

$$\begin{aligned} A &\rightarrow Bb \mid \dots \\ B &\rightarrow Aa \mid \dots \end{aligned}$$

中。这样的规则在真正的程序设计语言文法中几乎不可能发生，但本书为了完整仍给出它的解决方法。首先考虑直接左递归。

情况1：简单直接左递归

在这种情况下，左递归只出现在格式

$$A \rightarrow A \mid$$

的文法规则中，其中 \mid 和 ϵ 是终结符和非终结符的串，而且 A 不以 A 开头。在 3.2.3 节中，我们看到这个文法规则生成了格式 A^n ($n \geq 0$) 的串。选择 $A \rightarrow A$ 是基本情况，而 $A \rightarrow A \mid$ 是递归情况。

为了消除左递归，将这个文法规则重写为两个规则：一个是首先生成 ϵ ，另一个是生成 A 的重复，它不用左递归却用右递归：

$$\begin{aligned} A &\rightarrow A \\ A &\rightarrow A \mid \epsilon \end{aligned}$$

例4.1 再次考虑在简单表达式文法中的左递归规则：

$$\text{exp} \rightarrow \text{exp} \text{ addop term} \mid \text{term}$$

它属于格式 $A \rightarrow A \mid$ ，且 $A = \text{exp}$ ， $\mid = \text{addop term}$ ， $\epsilon = \text{term}$ 。将这个规则重写以消除左递归，就可得到

$$\begin{aligned} \text{exp} &\rightarrow \text{term exp} \\ \text{exp} &\rightarrow \text{addop term exp} \mid \epsilon \end{aligned}$$

情况2：普遍的直接左递归

这种情况发生在有如下格式的产生式

$$A \rightarrow A_1 \mid A_2 \mid \dots \mid A_n \mid A_1 A_2 \dots A_n \mid \dots \mid A_m$$

中。其中 A_1, \dots, A_m 均不以 A 开头。在这种情况下，其解法与简单情况类似，只需将选择相应地扩展：

$$\begin{aligned} A &\rightarrow A_1 A \mid A_2 A \mid \dots \mid A_m A \\ A &\rightarrow A_1 A \mid A_2 A \mid \dots \mid A_n A \mid \epsilon \end{aligned}$$

例4.2 考虑文法规则

$$\text{exp} \rightarrow \text{exp} + \text{term} \mid \text{exp} - \text{term} \mid \text{term}$$

如下消除左递归：

$$\begin{aligned} \text{exp} &\rightarrow \text{term exp} \\ \text{exp} &\rightarrow + \text{term exp} \mid - \text{term exp} \mid \epsilon \end{aligned}$$

情况3：一般的左递归

这里描述的算法仅是指不带有 ϵ 产生式且不带有循环的文法，其中循环 (cycle) 是至少有一步是以相同的非终结符： $A \xrightarrow{*} A$ 开始和结束的推导。循环几乎肯定能导致分析程序进入无穷循环，而且带有循环的文法从不作为程序设计语言文法出现。程序设计语言文法确实是有 ϵ 产生式，但这经常是在非常有限的格式中，所以这个算法对于这些文法也几乎总是有效的。

该算法的方法是：为语言的所有非终结符选择任意顺序，如 A_1, \dots, A_m ，接着再消除不增加 A_i 索引的所有左递归。它消除所有 $A_i \rightarrow A_j$ ，其中 $j \geq i$ 形式的规则。如果按这样操作从 1 到 m 的每一个 i ，则由于这样的循环的每个步骤只增加索引，且不能再次到达原始索引，因此就不会留下任何递归循环了。程序清单 4-3 是这个算法的详细步骤。

程序清单4-3 普遍左递归消除的算法

```

for  $i := 1$  to  $n$  do
  for  $j := 1$  to  $i-1$  do
    replace each grammar rule choice of the form  $A_i \rightarrow A_j \beta$  by the rule
       $A_i \rightarrow \alpha_1 \beta \mid \alpha_2 \beta \mid \dots \mid \alpha_k \beta$ , where  $A_j \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_k$  is
        the current rule for  $A_j$ 
    remove, if necessary, immediate left recursion involving  $A_i$ 

```

例4.3 考虑下面的文法：

$$\begin{aligned} A & \quad B a \mid A a \mid c \\ B & \quad B b \mid A b \mid d \end{aligned}$$

(由于该情况在任何标准程序设计语言中都不会发生，所以这个文法完全是自造的)。

为了使用算法，就须令 B 的数比 A 的数大（即 $A_1 = A$ ，且 $A_2 = B$ ）。因为 $n = 2$ ，则图4-3中的算法的外循环执行两次，一次是当 $i = 1$ ，另一次是当 $i = 2$ 。当 $i = 1$ 时，不执行内循环（带有索引 j ），这样唯一的动作就是消除 A 的直接左递归。最后得到的文法是

$$\begin{aligned} A & \quad B a A \mid c A \\ A & \quad a A \mid \varepsilon \\ B & \quad B b \mid A b \mid d \end{aligned}$$

现在为 $i = 2$ 执行外循环，且还执行一次内循环，此时 $j = 1$ 。在这种情况下，我们通过用第1个规则中的选择替换 A 而省略了规则 $B \rightarrow A b$ 。因此就得到了文法

$$\begin{aligned} A & \quad B a A \mid c A \\ A & \quad a A \mid \varepsilon \\ B & \quad B b \mid B a A b \mid c A b \mid d \end{aligned}$$

最后消除 B 的直接左递归以得到

$$\begin{aligned} A & \quad B a A \mid c A \\ A & \quad a A \mid \varepsilon \\ B & \quad c A b B \mid d B \\ B & \quad b B \mid a A b B \mid \varepsilon \end{aligned}$$

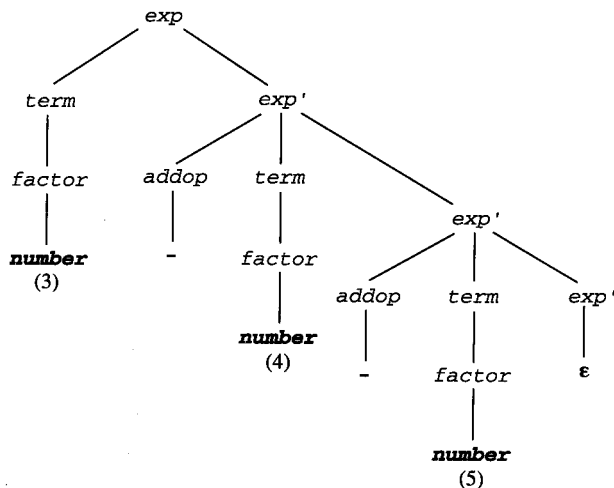
这个文法没有左递归。

左递归消除并不改变正被识别的语言，但它却改变了文法和分析树。这种改变确实导致了分析程序变得复杂起来（对于分析程序设计人员而言，也更困难了）。例如在前面作为标准示例的简单表达式文法中，该文法是左递归的，表达运算的结合性的表达式也是左递归。若要像在例4.1中一样消除直接左递归，就可得到图4-1中所给出的文法。

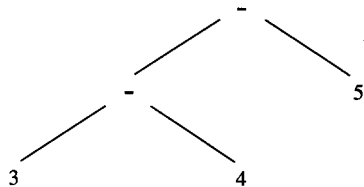
$$\begin{aligned} \text{exp} & \quad \text{term exp} \\ \text{exp} & \quad \text{addop term exp} \mid \varepsilon \\ \text{addop} & \quad + \mid - \\ \text{term} & \quad \text{factor term} \\ \text{term} & \quad \text{mulop factor term} \mid \varepsilon \\ \text{mulop} & \quad * \\ \text{factor} & \quad (\text{exp}) \mid \text{number} \end{aligned}$$

图4-1 消除了左递归的简单算术表达式文法

现在考虑表达式 3-4-5 的分析树：



这个树不再表达减法的左结合性了，然而，分析程序应仍构造出恰当的左结合语法树：



使用新文法来完成它不是完全微不足道的。为了看清原因，可先考虑利用给出的分析树来代替计算表达式的值这样略为简单的问题。为了做到这一点，必须将值 3 从根 *exp* 节点传送到它的右子节点 *exp*。之后，这个 *exp* 节点必须减去 4，并将新值 -1 向上传到它的最右边的子节点（另一个 *exp*）。这个 *exp* 节点按顺序也必须减去 5 并将值 -6 传送给最后的 *exp* 节点。这个节点仅有一个 ϵ 子节点，且它仅仅是将值 -6 传回来。接着，该值被向上返回到树的根 *exp* 节点，它就是表达式的最终值。

考虑它在递归下降分析程序中是如何工作的。其左递归消除的文法将产生如下过程 *exp* 和 *exp*：

```

procedure exp ;
begin
    term ;
    exp ;
end exp ;
procedure exp ;
begin
    case token of
      + : match ( + ) ;
          term ;
          exp ;
      - : match ( - ) ;

```

```

    term ;
    exp ;
end case ;
end exp ;

```

为了使这些过程真正计算表达式的值，应将其如下所示重写：

```

function exp : integer ;
var temp : integer ;
begin
    temp := term ;
    return exp (temp) ;
end exp ;

function exp ( valsofar : integer ) : integer ;
begin
    if token = + or token = - then
        case token of
            + : match ( + ) ;
                valsofar := valsofar + term ;
            - : match ( - ) ;
                valsofar := valsofar - term ;
        end case ;
        return exp (valsofar) ;
    else return valsofar ;
end exp ;

```

请注意 *exp* 过程现在是如何需要一个来自 *exp* 过程的参数。若这些过程将返回一个（左结合的）语法树，则会发生类似的情况。在 4.1 节里，给出的代码使用了基于 EBNF 的更为简单的解法中，它并不要求额外的参数。

最后，我们留意到程序清单 4-1 中的新表达式文法实际上是一个 LL(1) 文法。LL(1) 分析表在表 4-4 中给出了。正如对前面表格的处理一样，我们将在下一节再谈到它的构造。

表 4-4 程序清单 4-1 中文法的 LL(1) 分析表

$M[N, T]$	(number)	+	-	*	\$
<i>exp</i>	<i>exp</i> <i>term exp</i>	<i>exp</i> <i>term exp</i>					
<i>exp</i>			<i>exp</i> ϵ	<i>exp</i> <i>addop</i> <i>term exp</i>	<i>exp</i> <i>addop</i> <i>term exp</i>		<i>exp</i> ϵ
<i>addop</i>				<i>addop</i> +	<i>addop</i> -		

(续)

$M[N, T]$	(number)	+	-	*	\$
term	term factor term	term factor term					
term			term ϵ	term ϵ	term ϵ	term mulop factor term	term ϵ
mulop						mulop *	
factor	factor (exp)	factor number					

2) 提取左因子 当两个或更多文法规则选择共享一个通用前缀串时，需要提取左因子。如

$$A \rightarrow \mid$$

以下是语句序列的右递归示例（第3章的例3.7）：

$$\begin{aligned} \text{stmt-sequence} & \rightarrow \text{stmt} ; \text{stmt-sequence} \mid \text{stmt} \\ \text{stmt} & \rightarrow \mathbf{s} \end{aligned}$$

以下是if语句的随后版本：

$$\begin{aligned} \text{if-stmt} & \rightarrow \mathbf{if} (\text{exp}) \text{statement} \\ & \mid \mathbf{if} (\text{exp}) \text{statement} \mathbf{else} \text{statement} \end{aligned}$$

很明显，LL(1)分析程序不能区分这种情况中的产生式选择。这个简单情况的解法是将左边的分解出来，并将该规则重写为两个规则

$$\begin{aligned} A & \rightarrow A \\ A & \rightarrow \mid \end{aligned}$$

（如想用括号作为文法规则中的元符号，则也可写作 $A \rightarrow (\mid)$ ，它看起来很像算术中的因子分解）。为了使提取左因子能够正常进行，就必须确保 实际上是与右边共享的最长串。这里还可能享有相同前缀的超过两个的选择。程序清单 4-4中的是普通算法，之后还有一些示例。请注意，在处理算法时，每个共享相同前缀的非终结符的产生式的选择数目，每一步至少减少一个，这样算法才能够保证终止。

程序清单 4-4 提取左因子文法的算法

```

while there are changes to the language do
  for each nonterminal A do
    let  $\alpha$  be a prefix of maximal length that is shared
      by two or more production choices for A
    if  $\alpha \neq \epsilon$  then
      let  $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$  be all the production choices for A
      and suppose that  $\alpha_1, \dots, \alpha_k$  share  $\alpha$ , so that
       $A \rightarrow \alpha \beta_1 \mid \dots \mid \alpha \beta_k \mid \alpha_{k+1} \mid \dots \mid \alpha_n$ , the  $\beta_j$ 's share

```

no common prefix, and the $\alpha_{k+1}, \dots, \alpha_n$ do not share α
 replace the rule $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$ by the rules
 $A \rightarrow \alpha A' \mid \alpha_{k+1} \mid \dots \mid \alpha_n$
 $A' \rightarrow \beta_1 \mid \dots \mid \beta_k$

例4.4 考虑语句序列的文法，它写在右递归格式中就是：

$$\begin{aligned} \text{stmt-sequence} & \quad \text{stmt} ; \text{stmt-sequence} \mid \text{stmt} \\ \text{stmt} & \quad \mathbf{s} \end{aligned}$$

stmt-sequence的文法规则有一个共享的前缀，它可按如下提取左因子：

$$\begin{aligned} \text{stmt-sequence} & \quad \text{stmt stmt-seq} \\ \text{stmt-seq} & \quad ; \text{stmt-sequence} \mid \epsilon \end{aligned}$$

请注意，如果已经是左递归，而不是右递归地写出了 stmt-sequence 规则

$$\text{stmt-sequence} \quad \text{stmt-sequence} ; \text{stmt} \mid \text{stmt}$$

则消除直接左递归将会导致规则

$$\begin{aligned} \text{stmt-sequence} & \quad \text{stmt stmt-seq} \\ \text{stmt-seq} & \quad ; \text{stmt stmt-seq} \mid \epsilon \end{aligned}$$

这与从提取左因子中得到的结果几乎一样，在最后的规则中将 stmt stmt-seq 替换成 stmt-sequence 就使得两个结果一样。

例4.5 考虑if语句的如下（部分）文法：

$$\begin{aligned} \text{if-stmt} & \quad \mathbf{if} (\text{exp}) \text{statement} \\ & \quad \mid \mathbf{if} (\text{exp}) \text{statement} \mathbf{else} \text{statement} \end{aligned}$$

在这个文法中，提取了左因子的格式是

$$\begin{aligned} \text{if-stmt} & \quad \mathbf{if} (\text{exp}) \text{statement} \text{else-part} \\ \text{else-part} & \quad \mathbf{else} \text{statement} \mid \epsilon \end{aligned}$$

这正是在4.2.2节中所用到的格式（参见表4-2）。

例4.6 假设写出一个算术表达式文法，在其中给出了算术运算右结合性，而并非左结合性（这里用+仅是作为一个具体示例）：

$$\text{exp} \quad \text{term} + \text{exp} \mid \text{term}$$

这个文法需要提取左因子，则得到规则

$$\begin{aligned} \text{exp} & \quad \text{term exp} \\ \text{exp} & \quad + \text{exp} \mid \epsilon \end{aligned}$$

现在继续做例4.4，假设在第2个规则中将 exp 替换为 term exp（由于这个扩展无论如何都会在推导中的下一步发生，所以这是正规的）。接着就得到

$$\begin{aligned} \text{exp} & \quad \text{term exp} \\ \text{exp} & \quad + \text{term exp} \mid \epsilon \end{aligned}$$

这与通过消除左递归而从左递归规则中得到的文法相同。因此，提取左因子和消除左递归都可导致语言结构的语义晦涩（在这种情况下，它们都会阻碍结合性）。

例如，若要保存来自上面文法规则的右结合性（在别的格式中），就必须将每个+运算安排在结尾而不是开头。请读者为它写出一个递归下降程序过程。

例4.7 因为过程调用和赋值均以标识符开头，所以这是程序设计语言文法不能成为 LL(1)文法的典型情况。我们写出这个问题的以下表示：

```
statement  assign-stmt | call-stmt | other
assign-stmt  identifier := exp
call-stmt    identifier ( exp-list )
```

因为 **identifier** 作为 *assign-stmt* 和 *call-stmt* 共享的第1个记号，所以这个文法不是 LL(1)；而 **identifier** 也就是两个的先行记号。不幸的是，该文法不是位于一个可提取左因子的格式中。我们必须做的是首先将 *assign-stmt* 和 *call-stmt* 用它们的定义产生式的右边代替，如下：

```
statement  identifier := exp
           | identifier ( exp-list )
           | other
```

接着提取左因子以得到

```
statement  identifier statement
           | other
statement  := exp | ( exp-list )
```

请注意它如何通过从真正的调用或赋值动作（由 *statement* 代表）中分隔标识符（被赋予的值或被调用的过程）而使调用的语义和赋值变得晦涩。LL(1)分析程序必须通过使标识符以某种方式（如一个参数）对调用或赋值步骤有用，或调整语法树来弥补它。

最后，应指出在所有的例子中，我们已使提取左因子确实是在转换之后变成了 LL(1)文法。下一节将为其中的一些构造 LL(1)分析表，另一些则留在练习中。

4.2.4 在LL(1)分析中构造语法树

我们还需探讨 LL(1)分析如何适应于构造语法树而不是分析树（在 4.2.1 节中已描述了如何利用分析栈构造分析树）。我们看到有关适用于语法树构造方法的递归下降程序分析的章节相对简单，而 LL(1)分析程序却难以适用。其部分原因在于，正如我们所看到的，语法树的结构（如左结合性）会因提取左因子和消除左递归而变得晦涩。但是它的主要原因却是分析栈所代表的仅是预测的结构，而不是已经看到的结构。因此，语法树节点的构造必须推迟到将结构从分析栈中移走时，而不是当它们首次被压入时。一般而言，这就要求使用一个额外的栈来记录语法树节点，并在分析栈中放入“动作”标记来指出什么动作何时将在树栈中发生。自底向上分析程序（下一章）则易于适应使用了分析栈的语法树的构造，而且因此也就倾向于基于栈的表驱动分析方法。所以我们只给出一个简要的示例来解释它是如何用于 LL(1)分析的。

例4.8 我们使用一个仅带有一个加法运算地表达式文法。其 BNF 是

$$E \rightarrow E + n \mid n$$

这样就使得加法为左结合了。相应地带有消除左递归的 LL(1)文法是

$$\begin{aligned} E &\rightarrow n E \\ E &\rightarrow + n E \mid \varepsilon \end{aligned}$$

现在说明如何使用这个文法来计算表达式的算术值（语法树的构造是类似的）。

为了计算一个表达式的值，就要用一个单独的栈来存储计算的中间值，这个栈称作值栈（value stack）。在这个栈上必须安排两个运算。第1个运算是在输入中匹配数时将它压入，第2个是在栈中两数相加的运算。第1个运算由match过程完成（基于它匹配的记号），第2个需要被安排在分析栈上。它是通过将特殊符号压入分析栈中，而当其弹出时，则表示完成了一个加法运算。此处所用的符号是（#）。这个符号现在成为一个新的栈符号，而且必须也被添加到匹配一个“+”的文法规则中，即 E 规则：

$$E \rightarrow + n \# E \mid \varepsilon$$

请注意，将加法放在紧随下一个数之后的位置上，但是却是在处理任何其他的 E 非终结符之前。这就保证了是左结合。现在来看看如何计算表达式 $3+4+5$ 的值。如前所述指出分析栈、输入和动作，但却将值栈放在右边（它向左增长）。表4-5是分析程序的动作。

表4-5 例4.8的带有值栈动作的分析栈

分 析 栈	输 入	动 作	值 栈
$\$ E$	3 + 4 + 5	$E \rightarrow n E'$	\$
$\$ E n$	3 + 4 + 5	匹配/压入	\$
$\$ E'$	+ 4 + 5	$E' \rightarrow + n \# E'$	3 \$
$\$ E' \# n +$	+ 4 + 5	匹配	3 \$
$\$ E' \# n$	4 + 5	匹配/压入	3 \$
$\$ E' \#$	+ 5	加法栈	4 3 \$
$\$ E'$	+ 5	$E' \rightarrow + n \# E'$	7 \$
$\$ E' \# n +$	+ 5	匹配	7 \$
$\$ E' \# n$	5	匹配/压入	7 \$
$\$ E' \#$	\$	加法栈	5 7 \$
$\$ E'$	\$	$E' \rightarrow \varepsilon$	12 \$
\$	\$	接受	12 \$

请注意，当发生一个加法时，操作数按相反顺序位于栈中。这是这种基于栈的赋值的典型安排。

4.3 First集合和Follow集合

为了完成LL(1)分析算法，我们开发了一个构造LL(1)分析表的算法。正如早先已指出的，它涉及到计算First集合和Follow集合，本节将给出这些集合的定义和构造，之后再准确描述LL(1)分析表的构造。本节最后简要地介绍如何将该构造扩展为多于一个向前看符号。

4.3.1 First集合

定义：令 X 为一个文法符号（一个终结符或非终结符）或 ε ，则集合 $\text{First}(X)$ 由终结符组成，此外可能还有 ε ，它的定义如下：

1. 若 X 是终结符或 ε ，则 $\text{First}(X) = \{X\}$ 。
2. 若 X 是非终结符，则对于每个产生式 $X \rightarrow X_1 X_2 \dots X_n$ ， $\text{First}(X)$ 都包含了 $\text{First}(X_1) - \{\varepsilon\}$ 。若对于某个 $i < n$ ，所有的集合 $\text{First}(X_1), \dots, \text{First}(X_i)$ 都包括了

ϵ , 则 $\text{First}(X)$ 也包括了 $\text{First}(X_{i+1}) - \{\epsilon\}$ 。若所有集合 $\text{First}(X_1), \dots, \text{First}(X_n)$ 包括了 ϵ , 则 $\text{First}(X)$ 也包括 ϵ 。

现在为任意串 $X = X_1 X_2 \dots X_n$ (终结符和非终结符的串) 定义 $\text{First}(\alpha)$, 如下所示: $\text{First}(\alpha)$ 包括 $\text{First}(X_1) - \{\epsilon\}$ 。对于每个 $i = 2, \dots, n$, 如果对于所有的 $k = 1, \dots, i-1$, $\text{First}(X_k)$ 包括了 ϵ , 则 $\text{First}(\alpha)$ 就包括了 $\text{First}(X_i) - \{\epsilon\}$ 。最后, 如果对于所有的 $i = 1, \dots, n$, $\text{First}(X_i)$ 包括了 ϵ , 则 $\text{First}(\alpha)$ 也包括了 ϵ 。

这个定义可很容易地转化为一个算法。实际上, 唯一困难的是为每个非终结符 A 计算 $\text{First}(A)$, 这是因为终结符的 First 集合是很简单的, 并且串 X 的 First 集合从几乎 n 个单个符号的 First 集合建立, 其中 n 是在 X 中的符号数。因此只有在非终结符时才为算法写成伪代码, 如程序清单4-5所示。

程序清单4-5 为所有的非终结符 A 计算 $\text{First}(A)$ 的算法

```

for all nonterminals  $A$  do  $\text{First}(A) := \{\}$ ;
while there are changes to any  $\text{First}(A)$  do
  for each production choice  $A \rightarrow X_1 X_2 \dots X_n$  do
     $k := 1$ ;  $\text{Continue} := \text{true}$ ;
    while  $\text{Continue} = \text{true}$  and  $k \leq n$  do
      add  $\text{First}(X_k) - \{\epsilon\}$  to  $\text{First}(A)$ ;
      if  $\epsilon$  is not in  $\text{First}(X_k)$  then  $\text{Continue} := \text{false}$ ;
       $k := k + 1$ ;
    if  $\text{Continue} = \text{true}$  then add  $\epsilon$  to  $\text{First}(A)$ ;

```

也可以很容易看出如何在没有 ϵ 产生式的情况下解释这个定义: 只需不断为每个非终结符 A 和产生式选择 $A \rightarrow X_1 \dots$, 向 $\text{First}(A)$ 增加 $\text{First}(X_1)$ 一直到再没有增加什么。换言之, 只需考虑程序清单4-5中 $k = 1$ 的情况, 而不需要 **while** 内循环。我们将这个算法单独写在程序清单4-6中。当存在 ϵ 产生式时, 情况就复杂一些了, 这是因为必须查清 ϵ 是否在 $\text{First}(X_1)$ 中, 若是则为 X_2 继续相同的处理, 等等。该处理将一直延续到有穷步骤之后才结束; 但是实际上, 这个处理不但计算可作为从非终结符派生的串的第1个符号而出现的终结符, 而且它还决定非终结符是否可派生空串 (即: 消失)。这样的非终结符称为可空的:

程序清单4-6 当没有 ϵ 产生式时, 程序清单4-5的简化了的算法

```

for all nonterminals  $A$  do  $\text{First}(A) := \{\}$ ;
while there are changes to any  $\text{First}(A)$  do
  for each production choice  $A \rightarrow X_1 X_2 \dots X_n$  do
    add  $\text{First}(X_1)$  to  $\text{First}(A)$ ;

```

定义: 当存在一个推导 $A \xRightarrow{*} \epsilon$ 时, 非终结符 A 称作可空的 (nullable)。

现在给出以下的法则。

定理: 当且仅当 $\text{First}(A)$ 包含 ϵ 时, 非终结符 A 为可空的。

证明: 下面证明若 A 是可空的, 则 $\text{First}(A)$ 包含了 ϵ 。其逆命题也可用相似的方法得到证明。我们利用对产生式长度的归纳来推导。若 $A \xRightarrow{*} \epsilon$, 则必有一个产生式 $A \rightarrow \epsilon$, 且由定义可得 $\text{First}(A)$ 包含了 $\text{First}(\epsilon) = \{\epsilon\}$ 。现在假设对于长度 $< n$ 的推导的语句为真, 并令 $A \xRightarrow{*} X_1 \dots X_k \xRightarrow{*} \epsilon$ 是长度为 n 的一个推导 (利用产生式选择 $A \rightarrow X_1 \dots X_k$)。如果任何

一个 X_i 都是终结符,则它们都不能派生 ε ,所以所有的 X_i 都必须是非终结符。实际上,上面的推导并不完整,它意味着每个 $X_i \xrightarrow{*} \varepsilon$,且推导少于 n 个步骤。因此,通过归纳假设,对于每个 i , $\text{First}(X_i)$ 都包含了 ε 。最后,由定义可得 $\text{First}(A)$ 必须包含 ε 。

下面给出一些非终结符的First集合的计算示例。

例4.9 考虑简单整型表达式文法^①:

```

exp    exp addop term | term
addop  + | -
term    term mulop factor | factor
mulop  *
factor  ( exp ) | number

```

我们分别写出每个选择,这样就可按顺序思考它们了(还可为了引用作上编号):

- (1) $exp \rightarrow exp \text{ addop } term$
- (2) $exp \rightarrow term$
- (3) $addop \rightarrow +$
- (4) $addop \rightarrow -$
- (5) $term \rightarrow term \text{ mulop } factor$
- (6) $term \rightarrow factor$
- (7) $mulop \rightarrow *$
- (8) $factor \rightarrow (\text{ exp })$
- (9) $factor \rightarrow \text{ number}$

这个文法并未包括 ε 产生式,所以可使用程序清单4-6中简化了的算法。我们还注意到左递归规则1和5都未向First集合的计算增加任何东西^②。例如,文法规则1规定只有那个 $\text{First}(exp)$ 应被添加到 $\text{First}(exp)$ 中。因此,可从计算中删除这些产生式。但在这个例子中,为了表示清楚仍把它们保留在列表中。

现在应用程序清单4-6的算法,并同时按照刚才给出的顺序考虑产生式。产生式1未做任何变化。产生式2将 $\text{First}(term)$ 的内容增加到 $\text{First}(exp)$,但是 $\text{First}(term)$ 当前为空,所以它也没有任何变化。规则3和4分别给 $\text{First}(addop)$ 增添“+”和“-”,所以 $\text{First}(addop) = \{+, -\}$ 。规则5并未增加任何东西。规则6将 $\text{First}(factor)$ 增添到 $\text{First}(term)$ 中,但是 $\text{First}(factor)$ 当前仍为空,所以也未发生任何改变。规则7向 $\text{First}(mulop)$ 增添*,所以 $\text{First}(mulop) = \{*\}$ 。规则8将(增加到 $\text{First}(factor)$ 中,而规则9将**number**增加到 $\text{First}(factor)$ 中,所以 $\text{First}(factor) = \{ (, \text{ number} \}$ 。现在再从规则1开始,这是由于它并未有任何变化。现在规则1到5都未有任何变化($\text{First}(term)$ 仍为空)。规则6将 $\text{First}(factor)$ 增加到 $\text{First}(term)$ 中,且 $\text{First}(factor) = \{ (, \text{ number} \}$,所以现在也有 $\text{First}(term) = \{ (, \text{ number} \}$ 。规则8和9未带来任何变化。我们必须再一次从规则1开始,这是因为有一个集合改变了,规则2最后将 $\text{First}(term)$ 的内容增加到 $\text{First}(exp)$ 中,且 $\text{First}(exp) = \{ (, \text{ number} \}$ 。需要使用多遍文法规则,直到再没有改变发生,所以在4遍之后,

① 这个文法具有左递归但不是LL(1),所以我们不能为它建立一个LL(1)分析表。但是它仍是一个对于解释如何计算First集合的有用示例。

② 当存在 ε 产生式时,左递归规则可用于First集合。

就已计算出以下的First集合：

$$\begin{aligned}\text{First}(exp) &= \{ (, \text{number} \} \\ \text{First}(term) &= \{ (, \text{number} \} \\ \text{First}(factor) &= \{ (, \text{number} \} \\ \text{First}(addop) &= \{ +, - \} \\ \text{First}(mulop) &= \{ * \}\end{aligned}$$

(请注意，如果是一开始而不是最后为 *factor* 列出文法规则，则可将文法规则使用遍数由 4 减少为 2)。表 4-6 列出了这个计算。在这个表中，只是在它们发生的恰当位置中将记录改变。空的项目表示在该步骤中串没有发生变化。由于没有改变发生，所以最后一遍不再进行。

表 4-6 为例 4.9 中的文法计算 First 集合

文 法 规 则	第 1 遍	第 2 遍	第 3 遍
<i>exp</i> <i>exp</i> <i>addop term</i> <i>exp</i> <i>term</i>			$\text{First}(exp) =$ $\{ (, \text{number} \}$
<i>addop</i> +	$\text{First}(addop)$ $= \{ + \}$		
<i>addop</i> -	$\text{First}(addop)$ $= \{ +, - \}$		
<i>term</i> <i>term</i> <i>mulop factor</i> <i>term</i> <i>factor</i>		$\text{First}(term) =$ $\{ (, \text{number} \}$	
<i>mulop</i> *	$\text{First}(mulop)$ $= \{ * \}$		
<i>factor</i> (exp)	$\text{First}(factor)$ $= \{ (\}$		
<i>factor</i> number	$\text{First}(factor) =$ $\{ (, \text{number} \}$		

例 4.10 考虑 if 语句 (例 4.5) 的 (提取左因子) 文法：

$$\begin{aligned}\text{statement} & \quad \text{if-stmt} \mid \text{other} \\ \text{if-stmt} & \quad \text{if} (exp) \text{statement else-part} \\ \text{else-part} & \quad \text{else statement} \mid \varepsilon \\ \text{exp} & \quad 0 \mid 1\end{aligned}$$

这个文法确实有 ε 产生式，但只有 *else-part* 非终结符是可空的，所以计算的难度最小。由于没有一个是以其 First 集合包括了 ε 的非终结符开头，所以其实只需要在某一步中增加 ε ，并保持其他步骤不受影响。这对于真正的程序设计语言文法是非常典型的，其中的 ε 产生式几乎总是非常有限的，且极少显示在普通情况中的复杂性。

同前面的一样，单独写出各个文法规则选择，并给其编号：

- (1) *statement* *if-stmt*
- (2) *statement* **other**
- (3) *if-stmt* **if** (*exp*) *statement else-part*
- (4) *else-part* **else** *statement*
- (5) *else-part* ϵ
- (6) *exp* **0**
- (7) *exp* **1**

我们再次一个一个地运行产生式选择的步骤，一旦一个 First 集合在前一遍中有了改变，就制造出新的一遍来。因为 First (*if-stmt*) 为空，所以文法规则 1 开始时没有变化。规则 2 将终结符 **other** 增加到 First (*if-stmt*) 中，所以 First (*statement*) = { **other** }。规则 3 将 **if** 增加到 First (*if-stmt*) 中，所以 First (*if-stmt*) = { **if** }。规则 4 在 First (*else-part*) 中增加 **else**，所以 First (*else-part*) = { **else** }。规则 5 在 First (*else-part*) 中增加 ϵ ，所以 First (*else-part*) = { **else**, ϵ }。规则 6 和规则 7 分别将 **0** 和 **1** 增加到 First (*exp*) 中，所以 First (*exp*) = { **0**, **1** }。现在从规则 1 开始做另一遍。由于 First (*if-stmt*) 包括了 **if** 终结符，所以现在该规则又将 **if** 增加到 First (*statement*) 中。由此，First (*statement*) = { **if**, **other** }。第 2 遍中再没有其他改变了，且第 3 遍也无任何改变。因此，我们计算出以下 First 集合：

First (*statement*) = { **if**, **other** }
 First (*if-stmt*) = { **if** }
 First (*else-part*) = { **else**, ϵ }
 First (*exp*) = { **0**, **1** }

表 4-7 以与表 4-6 类似的方式展示了这个计算。同前面一样，该表只显示改变，且不显示最后一遍（因为在其中没有改变）。

表 4-7 为例 4.10 中的文法计算 First 集合

文法规则	第 1 遍	第 2 遍
<i>statement</i> <i>if-stmt</i>		First (<i>statement</i>) = { if , other }
<i>statement</i> other	First (<i>statement</i>) = { other }	
<i>if-stmt</i> if (<i>exp</i>) <i>statement else-part</i>	First (<i>if-stmt</i>) = { if }	
<i>else-part</i> else <i>statement</i>	First (<i>else-part</i>) = { else }	
<i>else-part</i> ϵ	First (<i>else-part</i>) = { else , ϵ }	
<i>exp</i> 0	First (<i>exp</i>) = { 0 }	
<i>exp</i> 1	First (<i>exp</i>) = { 0 , 1 }	

例 4.11 考虑下面的语句序列文法（参见例 4.4）：

stmt-sequence *stmt stmt-seq*
stmt-seq' ; *stmt-sequence* | ϵ

$$stmt \quad s$$

我们再一次单独列出每个产生式选择：

- (1) $stmt-sequence \quad stmt \, stmt-seq$
- (2) $stmt-seq \quad ; \, stmt-sequence$
- (3) $stmt-seq \quad \epsilon$
- (4) $stmt \quad s$

在第1遍中，规则1并未增加任何东西。规则2和规则3导致 $First(stmt-seq) = \{ ;, \epsilon \}$ 。规则4导致 $First(stmt) = \{ s \}$ 。在第2遍中，规则1这次使 $First(stmt-sequence) = First(stmt) = \{ s \}$ 。除此之外就再也没有其他改变了。第3遍也没有任何改变。我们计算出以下的First集合：

$$First(stmt-sequence) = \{ s \}$$

$$First(stmt) = \{ s \}$$

$$First(stmt-seq) = \{ ;, \epsilon \}$$

请读者自己构造一个与表4-6和表4-7类似的表格。

4.3.2 Follow 集合

定义：给出一个非终结符 A ，那么集合 $Follow(A)$ 则是由终结符组成，此外可能还有 $\$$ 。

集合 $Follow(A)$ 的定义如下：

1. 若 A 是开始符号，则 $\$$ 就在 $Follow(A)$ 中。
2. 若存在产生式 $B \rightarrow A \dots$ ，则 $First(\dots) - \{ \epsilon \}$ 在 $Follow(A)$ 中。
3. 若存在产生式 $B \rightarrow A \dots$ ，且 ϵ 在 $First(\dots)$ 中，则 $Follow(A)$ 包括 $Follow(B)$ 。

首先检查这个定义的内容，之后为由此引出的Follow集合的计算写出算法。读者首先应注意用作标记输入结束的“ $\$$ ”，它就像是Follow集合计算中的一个记号。若没有它，那么在整个被匹配的串之后就没有符号了。由于这样的串是由文法的开始符号生成的，所以 $\$$ 必须总是要增加到开始符号的Follow集合中（若开始符号从不出现在产生式的右边，那么它就是开始符号的Follow集合的唯一成分）。

读者其次需要注意的是：空的“伪记号” ϵ 永远也不是Follow集合的元素，它之所以有意义是因为 ϵ 在First集合中是被仅仅用来标记那些可消失的串。在输入中它不能真正地被识别出来。而另一方面，Follow符号则总是相对于现存的输入（包括 $\$$ 符号，它将匹配由扫描程序生成的EOF）来匹配的。

大家还应注意Follow集合仅是针对非终结符定义的，然而First集合却还可为终结符以及终结字符串和非终结字符串定义。我们可将Follow集合的定义扩展到符号串，但由于在构造LL(1)分析表时，仅需要非终结符的Follow集合，所以这是不必要的。

最后，我们还要留意Follow集合的定义在产生式的“右边”起作用，而First集合的定义却是在“左边”起作用。由此就可说若 A 不包括在产生式 $B \rightarrow \dots A \dots$ 中，则产生式 $B \rightarrow \dots A \dots$ 就没有任何有关 A 的Follow集合的信息。只有当 A 出现在产生式的右边时，才可得到 $Follow(A)$ 。所以一般地，每个文法规则选择都可得到右边的每个非终结符的Follow集合。与在First集合中的情况不同，每个文法规则选择添加唯一的非终结符（在左边的那个）的First集合。

此外，假设有文法规则 $A \rightarrow B \dots$ ，那么 $Follow(B)$ 将通过定义中的情况(3)包含 $Follow(A)$ 。这是因为在任何包括了 A 的串中， A 可被 B 代替（这是动作中的“上下文无关”）。这个特性在

某种意义上与First集合的情况相反：若有 $A \rightarrow B$ ，那么First(A)就包括了First(B)（除了 ϵ 的可能）。

程序清单4-7给出了计算由Follow集合的定义得出的Follow集合的算法。利用这个算法为相同的3个文法计算Follow集合，我们曾在前面为这3个文法计算了First集合（同在First集合中一样，当没有 ϵ 产生式时，算法就简化了，我们把它留给读者）。

程序清单4-7 计算Follow集合的算法

```

Follow(start-symbol) := { $ };
for all nonterminals A ≠ start-symbol do Follow(A) := { };
while there are changes to any Follow sets do
  for each production A → X1X2...Xn do
    for each Xi that is a nonterminal do
      add First(Xi+1Xi+2...Xn) - { ε } to Follow(Xi)
      (* Note: if i=n, then Xi+1Xi+2...Xn = ε *)
      if ε is in First(Xi+1Xi+2...Xn) then
        add Follow(A) to Follow(Xi)

```

例4.12 我们再考虑一下在例4.9中曾计算过的First集合的简单表达式文法，如下所示：

First(*exp*) = { (, **number** }
 First(*term*) = { (, **number** }
 First(*factor*) = { (, **number** }
 First(*addop*) = { +, - }
 First(*mulop*) = { * }

再次写出带有编号的产生式：

(1) *exp* → *exp addop term*
 (2) *exp* → *term*
 (3) *addop* → +
 (4) *addop* → -
 (5) *term* → *term mulop factor*
 (6) *term* → *factor*
 (7) *mulop* → *
 (8) *factor* → (*exp*)
 (9) *factor* → **number**

规则3、规则4、规则7和规则9的右边均无非终结符，所以它们未向Follow集合的计算增加任何东西。我们再按顺序考虑其他规则。在开始之前，先设Follow(*exp*) = { \$ }；其他的Follow集合都先设置为空。

规则1影响了3个非终结符的Follow集合：*exp*、*addop*和*term*。将First(*addop*)增加到Follow(*exp*)，所以现在Follow(*exp*) = { \$, +, - }。接着，将First(*term*)增加到Follow(*addop*)中，所以现在Follow(*addop*) = { (, **number** }。最后，将Follow(*exp*)增加到Follow(*term*)中，所以Follow(*term*) = { \$, +, - }。

规则2再次导致Follow(*exp*)被添加到Follow(*term*)中，但是这刚才已由规则1完成，所以Follow集合就不再发生什么改变了。

规则5有3个影响：将First (*mulop*)添加到Follow (*term*)中，所以Follow (*term*) = { $\$, +, -, *$ }。接着，将First (*factor*)被添加到Follow (*mulop*)中，所以Follow (*mulop*) = { $(, \textbf{number}$ }。最后将Follow (*term*)增加到Follow (*factor*)中，所以Follow (*factor*) = { $\$, +, -, *$ }。

规则6导致的结果与规则5的最后一步相同，所以也就没有任何改变了。

最后，规则8将First ($) = \{ \}$ 增加到Follow (*exp*)中，所以Follow (*exp*) = { $\$, +, -,)$ }。

在第2遍中，规则1将 $)$ 增加到Follow (*term*)中(所以Follow (*term*) = { $\$, +, -, *,)$ }；规则5将 $)$ 增加到Follow (*factor*)中(所以Follow (*factor*) = { $\$, +, -, *,)$ }。第3遍未引起任何改变，所以算法结束。这样我们就计算了以下的Follow集合：

Follow (*exp*) = { $\$, +, -,)$ }
 Follow (*addop*) = { $(, \textbf{number}$ }
 Follow (*term*) = { $\$, +, -, *,)$ }
 Follow (*mulop*) = { $(, \textbf{number}$ }
 Follow (*factor*) = { $\$, +, -, *,)$ }

同在计算First集合中的一样，我们将计算过程放在表 4-8中。同前面一样，在该表中省略了结束的遍而只指出当改变发生时 Follow集合的变化。我们还省略了对计算不可能有影响的 4个文法规则选择（之所以包括了两个规则 *exp term*和*term factor*，是因为它们虽然没有真正的影响，但仍存在可能的影响）。

表4-8 为例4.12中的文法计算Follow集合

文 法 规 则	第 1 遍	第 2 遍
<i>exp</i> <i>exp addop</i>	Follow (<i>exp</i>) =	Follow (<i>term</i>) =
<i>term</i>	{ $\$, +, -$ }	{ $\$, +, -, *,)$ }
	Follow (<i>addop</i>) =	
	{ $(, \textbf{number}$ }	
	Follow (<i>term</i>) =	
	{ $\$, +, -$ }	
<i>exp</i> <i>term</i>		
<i>term</i> <i>term mulop</i>	Follow (<i>term</i>) =	Follow (<i>factor</i>) =
<i>factor</i>	{ $\$, +, -, *$ }	{ $\$, +, -, *,)$ }
	Follow (<i>mulop</i>) =	
	{ $(, \textbf{number}$ }	
	Follow (<i>factor</i>) =	
	{ $\$, +, -, *$ }	
<i>term</i> <i>factor</i>		
<i>factor</i> (<i>exp</i>)	Follow (<i>exp</i>) =	
	{ $\$, +, -,)$ }	

例4.13 再次考虑if语句简化了的文法，在例4.10中已计算了它的First集合，如下：

First (*statement*) = { **if**, **other** }
 First (*if-stmt*) = { **if** }
 First (*else-part*) = { **else**, ϵ }

$$\text{First}(exp) = \{0, 1\}$$

这里再重复一下带有编号的产生式：

- (1) *statement* *if-stmt*
- (2) *statement* **other**
- (3) *if-stmt* **if** (*exp*) *statement* *else-part*
- (4) *else-part* **else** *statement*
- (5) *else-part* ϵ
- (6) *exp* 0
- (7) *exp* 1

规则2、规则5、规则6和规则7对Follow集合的计算没有影响，所以忽略它们。

首先设Follow(*statement*) = { \$ }，并将其他非终结符的Follow集合初始化为空。规则1现在将Follow(*statement*) 添加到Follow(*if-stmt*)中，所以Follow(*if-stmt*) = { \$ }。规则3影响到了*exp*、*statement*和*else-part*的Follow集合。首先，Follow(*exp*)得到First() = { }，所以Follow(*exp*) = { }。接着，Follow(*statement*) 得到First(*else-part*) - { ϵ }，所以Follow(*statement*) = { \$, **else** }。最后，将Follow(*if-stmt*) 添加到Follow(*else-part*) 和Follow(*statement*) 中（这是因为*if-stmt*会消失）。第1个加法得到了Follow(*else-part*) = { \$ }，但第2个却无任何变化。最后，规则4将Follow(*else-part*)添加到Follow(*statement*)中，同样也没有任何改变。

在第2遍中，规则1再次将Follow(*statement*)添加到Follow(*if-stmt*)中，导致了Follow(*if-stmt*) = { \$, **else** }。规则3现在将终结符**else**增加到Follow(*else-part*)中，所以Follow(*else-part*) = { \$, **else** }。最后，规则4并未带来任何改变。第3遍同样也无任何变化，所以计算出以下的Follow集合：

$$\begin{aligned}\text{Follow}(\textit{statement}) &= \{\$, \mathbf{else}\} \\ \text{Follow}(\textit{if-stmt}) &= \{\$, \mathbf{else}\} \\ \text{Follow}(\textit{else-part}) &= \{\$, \mathbf{else}\} \\ \text{Follow}(\textit{exp}) &= \{\}\end{aligned}$$

请读者像上例一样为这个计算构造一个表。

例4.14 为例4.11中简化的语句序列文法（并且带有文法规则选择）计算Follow集合：

- (1) *stmt-sequence* *stmt* *stmt-seq*
- (2) *stmt-seq* ; *stmt-sequence*
- (3) *stmt-seq* ϵ
- (4) *stmt* **s**

在例4.11中，计算了以下的First集合：

$$\begin{aligned}\text{First}(\textit{stmt-sequence}) &= \{\mathbf{s}\} \\ \text{First}(\textit{stmt}) &= \{\mathbf{s}\} \\ \text{First}(\textit{stmt-seq}) &= \{;, \epsilon\}\end{aligned}$$

文法规则3和规则4并未对Follow集合的计算有任何影响。我们从Follow(*stmt-sequence*) = { \$ }开始，并使其他Follow集合为空。规则1导致了Follow(*stmt*) = { ; }和Follow(*stmt-seq*) = { \$ }。规则2未带来任何影响。另一遍也未产生更多的改变。所以计算出Follow集合

$$\text{Follow}(\textit{stmt-sequence}) = \{\$ \}$$

$$\text{Follow}(stmt) = \{ ; \}$$

$$\text{Follow}(stmt-seq) = \{ \$ \}$$

4.3.3 构造LL(1)分析表

现在考虑LL(1)分析表中各项的初始结构，如在4.2.2节中给出的：

1) 如果 $A \rightarrow \alpha$ 是一个产生式选择，且有推导 $S \xRightarrow{*} \alpha A$ 成立，其中 S 是一个记号，就将 A 添加到表项 $M[A, a]$ 中。

2) 如果 $A \rightarrow \epsilon$ 是 ϵ 产生式，且有推导 $S \xRightarrow{*} \alpha A$ 成立，其中 S 是开始符号， a 是一个记号（或 $\$$ ），就将 $A \rightarrow \epsilon$ 添加到表项 $M[A, a]$ 中。

规则1中的记号 a 很明显是在 $\text{First}(\alpha)$ 中，且规则2的记号 a 是在 $\text{Follow}(A)$ 中，因此，就可得到LL(1)分析表的以下算法构造：

LL(1)分析表 $M[N, T]$ 的构造：为每个非终结符 A 和产生式 $A \rightarrow \alpha$ 重复以下两个步骤：

1) 对于 $\text{First}(\alpha)$ 中的每个记号 a ，都将 $A \rightarrow \alpha$ 添加到项目 $M[A, a]$ 中。

2) 若 ϵ 在 $\text{First}(\alpha)$ 中，则对于 $\text{Follow}(A)$ 的每个元素 a （记号或是 $\$$ ），都将 $A \rightarrow \epsilon$ 添加到 $M[A, a]$ 中。

下面的定理基本上是LL(1)文法的定义和刚才给出的分析表构造的直接结果，它的证明留在了练习中：

定理：若满足以下条件，则BNF中的文法就是LL(1)文法（LL(1) grammar）。

1. 在每个产生式 $A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$ 中，对于所有的 i 和 j ： $1 \leq i, j \leq n, i \neq j$ ， $\text{First}(\alpha_i) \cap \text{First}(\alpha_j) = \emptyset$ 。
2. 若对于每个非终结符 A 都有 $\text{First}(A) \cap \text{Follow}(A) = \emptyset$ ，那么 $\text{First}(A) \cap \text{Follow}(A) = \emptyset$ 。

现在针对前面的文法来看一些分析表的例子。

例4.15 考虑在本章中一直作为标准示例的简单表达式文法。该文法如最开始给出的（参见例4.9）一样为左递归。在前一节用消除左递归写出了一个相等的文法，如下：

```

exp  term exp
exp  addop term exp | ε
addop + | -
term  factor term
term  mulop factor term | ε
mulop *
factor ( exp ) | number

```

必须为这个文法的非终结符计算 First 集合和 Follow 集合。我们把这个计算留到了练习中，这里只是给出结果：

$$\text{First}(exp) = \{ (, \text{number} \}$$

$$\text{First}(exp) = \{ +, -, \epsilon \}$$

$$\text{First}(addop) = \{ +, - \}$$

$$\text{First}(term) = \{ (, \text{number} \}$$

$$\text{First}(term) = \{ *, \epsilon \}$$

$$\text{First}(mulop) = \{ * \}$$

$$\text{Follow}(exp) = \{ \$,) \}$$

$$\text{Follow}(exp) = \{ \$,) \}$$

$$\text{Follow}(addop) = \{ (, \text{number} \}$$

$$\text{Follow}(term) = \{ \$,), +, - \}$$

$$\text{Follow}(term) = \{ \$,), +, - \}$$

$$\text{Follow}(mulop) = \{ (, \text{number} \}$$

First (*factor*) = { (, **number** } Follow (*factor*) = { \$,), +, -, * }

应用刚才描述的 LL(1) 分析表就可得到如表 4-4 的表格了。

例 4.16 考虑 if 语句的简化了的文法：

```

statement  if-stmt | other
if-stmt    if ( exp ) statement else-part
else-part  else statement | ε
exp        0 | 1

```

该文法的 First 集合和 Follow 集合分别在例 4.10 和例 4.13 中已做过了计算。这里再把它列出来：

First (*statement*) = { **if**, **other** } Follow (*statement*) = { \$, **else** }
 First (*if-stmt*) = { **if** } Follow (*if-stmt*) = { \$, **else** }
 First (*else-part*) = { **else**, ε } Follow (*else-part*) = { \$, **else** }
 First (*exp*) = { 0, 1 } Follow (*exp*) = {) }

构造出的 LL(1) 分析表在表 4-3 中。

例 4.17 考虑例 4.4 中的文法（带有了提取左因子）：

```

stmt-sequence  stmt stmt-seq
stmt-seq       ; stmt-sequence | ε
stmt           s

```

该文法有以下的 First 集合和 Follow 集合：

First (*stmt-sequence*) = { **s** } Follow (*stmt-sequence*) = { \$ }
 First (*stmt*) = { **s** } Follow (*stmt*) = { ;, \$ }
 First (*stmt-seq*) = { ;, ε } Follow (*stmt-seq*) = { \$ }

下面是 LL(1) 分析表。

$M[N, T]$	s	;	\$
<i>stmt-sequence</i>	<i>stmt-sequence</i>		
	<i>stmt stmt-seq</i>		
<i>stmt</i>	<i>stmt</i> s		
<i>stmt-seq</i>		<i>stmt-seq</i>	<i>stmt-seq</i> ε
		<i>;</i> <i>stmt-sequence</i>	

4.3.4 再向前：LL(k) 分析程序

前面的工作可推广到先行 k 个符号。例如，可定义 $\text{First}_k() = \{ w_k \mid * w \}$ ，其中 w 是一个记号串，且 $w_k = w$ 的前 k 个记号（或若 w 中的记号少于 k 个，则为 w ）。类似地，还可定义 $\text{Follow}_k(A) = \{ w_k \mid S\$ * Aw \}$ 。虽然这些定义比起当 $k = 1$ 时的定义缺少一些“算法性”，但是仍可开发出计算这些集合的算法来，且 LL(k) 分析表的构造也可像前面一样完成。

在 LL(k) 分析中仍出现了一些复杂之处。首先，分析表变得大了许多，这是由于列数按 k 的指数次增加（对于一定的推广，可利用表压缩方法算出来）。其次，分析表本身并不表达 LL(k) 分析的全部能力，这主要是由于所有的 Follow 串并不在所有的上下文中发生。因此，使

用如前所构造的表的分析与 $LL(k)$ 分析就不同了, $LL(k)$ 分析被称作强 $LL(k)$ 分析(Strong $LL(k)$ parsing)或 $SLL(k)$ 分析($SLL(k)$ parsing)。读者可查阅“注意与参考”一节以得到更多的信息。

$LL(k)$ 分析程序和 $SLL(k)$ 分析程序在使用中都不普遍,其部分原因在于它们增加了难度,但主要原因还是在于对于任何的 k 而言,在 $LL(1)$ 中失败的文法在实际应用中也不可能会有 $LL(k)$ 。例如,无论 k 为多大,带有左递归的文法永远不会是 $LL(k)$ 。但是,递归下降分析程序却能在需要的时候选择使用更大的先行,甚至于如前所看到的对于任何 k ,可使用特别的方法来分析不是 $LL(k)$ 的文法。

4.4 TINY语言的递归下降分析程序

本节将讨论附录B中列出的TINY语言的完整的递归下降程序。分析程序构造了如上一章的3.7节所描述的语法树,除此之外,它还将语法树的表示打印到列表文件中。分析程序使用如程序清单4-8中所给出的EBNF,它与第3章的程序清单3-1相对应。

程序清单 4-8 EBNF中TINY语言的文法

```

program → stmt-sequence
stmt-sequence → statement { ; statement }
statement → if-stmt | repeat-stmt | assign-stmt | read-stmt | write-stmt
if-stmt → if exp then stmt-sequence [ else stmt-sequence ] end
repeat-stmt → repeat stmt-sequence until exp
assign-stmt → identifier := exp
read-stmt → read identifier
write-stmt → write exp
exp → simple-exp [ comparison-op simple-exp ]
comparison-op → < | =
simple-exp → term { addop term }
addop → + | -
term → factor { mulop factor }
mulop → * | /
factor → ( exp ) | number | identifier

```

TINY分析程序完全按照4.1节给出的递归下降程序的要点。这个分析程序包括两个代码文件：`parse.h`和`parse.c`。`parse.h`文件（附录B的第850行到第865行）极为简单：它由一个声明

```
TreeNode * parse (void);
```

组成,它定义了主分析例程`parse`,`parse`又返回一个指向由分析程序构造的语法树的指针。`parse.c`文件在附录B的第900行到第1114行中,它由11个相互递归的过程组成,这些过程与程序清单4-8中的EBNF文法直接对应:一个对应于`stmt-sequence`,一个对应于`statement`,5个分别对应于5种不同的语句,另有4个对应于表达式的不同优先层次。操作符非终结符并未包括到过程之中,但却作为与它们相关的表达式被识别。这里也没有过程与`program`相对应,这是因为一个程序就是一个语句序列,所以`parse`例程仅调用`stmt_sequence`。

分析程序代码还包括了一个保存向前看记号的静态变量`token`以及一个查找特殊记号的`match`过程,当它找到匹配时就调用`getToken`,否则就声明出错;此外代码还包括将出错信息打印到列表文件中的`syntaxError`过程。主要的`parse`过程将`token`初始化到输入的第1个记号中,同时调用`stmt_sequence`,接着再在返回由`stmt_sequence`构造的树之前检查源文件的末尾(如果在`stmt_sequence`返回之后还有更多的记号,那么这就是一个错误)。

每个递归过程的内容应有相对的自身解释性，`stmt_sequence`却有可能是一个例外，它可写在一个更为复杂的格式中以提高对出错的处理能力；本书将在出错处理的讨论中简要地解释这一点。递归分析过程使用 3 种实用程序过程，为了方便已将它们都放在了文件 `util.c`（附录 B 的第 350 行到第 526 行）中，此外它还带有接口 `util.h`（附录 B 的第 300 行到第 335 行）。这些过程是：

1) `newStmtNode`（第 405 行到第 421 行），它取用一个指示语句种类的参数，它还在分配该类的新语句节点的同时返回一个指向新分配的节点的指针。

2) `newExpNode`（第 423 行到第 440 行），它取用一个指示表达式种类的参数，它还在分配该类新表达式节点的同时返回一个指向新分配的节点的指针。

3) `copyString`（第 442 行到第 455 行），它取用一个串参数，为拷贝分配足够的空间，并拷贝串，同时返回一个指向新分配的拷贝的指针。

由于 C 语言不为串自动分配空间，且扫描程序会为它所识别的记号的串值（或词法）重复使用相同的空间，所以 `copyString` 过程是必要的。

`util.c` 中也包括了过程 `printTree`（第 473 行到第 506 行），`util.c` 将语法树的线性版本写在了列表中，这样就能看到分析的结果了。这个过程在全程变量 `traceParse` 控制之下从主程序中调用。

通过打印节点信息以及在此之后缩排到孩子信息中来操作 `printTree` 过程；从这个缩排中可构造真正的树。样本 TINY 程序的语法树（参见第 3 章的程序清单 3-3 和图 3-6）由 `traceParse` 打印到列表文件中，如程序清单 4-9 所示。

程序清单 4-9 由 `printTree` 过程显示一个 TINY 语法树

```
Read: x
If
  Op: <
    const: 0
    Id: x
  Assign to: fact
    const: 1
  Repeat
    Assign to: fact
      Op: *
        Id: fact
        Id: x
    Assign to: x
      Op: -
        Id: x
        const: 1
    Op: =
      Id: x
      const: 0
  Write
    Id: fact
```

4.5 自顶向下分析程序中的错误校正

分析程序对于语法错误的反应通常是编译器使用中的主要问题。在最低限度之下，分析程序应能判断出一个程序在语句构成上是否正确。完成这项任务的分析程序被称作识别程序

(recognizer), 这是因为它的工作是在由正讨论的程序设计语言生成的上下文无关语言中识别串。值得一提的是: 任何分析至少必须工作得像一个识别程序一样, 即: 若程序包括了一个语法错误, 则分析必须指出某个错误的存在; 反之若程序中没有语法错误, 则分析程序不应声称有错误存在。

除了这个最低要求之外, 分析程序可以显示出对于不同层次的错误的反应。通常地, 分析程序会试图给出一个有意义的错误信息, 这至少是针对于遇到的第 1 个错误, 此外它还试图尽可能地判断出错误发生的位置。一些分析程序甚至还可尝试着进行错误校正 (error correction) (或可能更恰当一些, 应说是错误修复 (error repair)), 在这里分析程序试图从给出的不正确的程序中推断出正确的程序。若它能这样做, 那么这种情况绝大多数遇到的都是简单问题, 如缺少了标点符号。在这里存在着一个算法体, 它可被用来寻找在某种意义上与已给出的程序 (通常是指必须被插入的、被删除的或被改变的记号) 最近的正确程序。这样的最小距离错误校正 (minimal distance error correction) 当用于每个错误时通常效率就太低了, 而且不管怎样它也是与程序员真正希望的相去甚远。因此, 在真正的分析程序中极少能看到它。编译器编写者发现若不做错误校正则很难生成有意义的错误信息。

用于错误校正的大多数技术都很特别, 在某种意义上说, 它们需要特殊的语言和特殊的分析算法, 在许多特殊情况下还要求有单独的解法, 对此很难得到一般原理。以下是一些重要的事项。

1) 分析程序应试着尽可能快地判断出错误的发生。在声明错误之前等待太久就意味着真正的错误的位置可能已丢失了。

2) 在错误发生之后, 分析程序必须挑选出一个可能的位置来恢复分析。分析程序应总是尝试尽可能多地分析代码, 这是为了在翻译中尽可能多地找到真实的错误。

3) 分析程序应避免出现错误级联问题 (error cascade problem), 在这里错误会产生一个冗长的虚假的出错信息。

4) 分析程序必须避免错误的无限循环, 此时不用任何输入都会产生一个出错信息的无限级联。

其中的一些目标自相矛盾, 所以编译器的编写者必须在构造错误处理器时应作一些折衷。例如, 避免错误级联和无穷循环问题会引起分析程序跳过一些输入, 它与处理尽可能多的输入的目标相折衷。

4.5.1 在递归下降分析程序中的错误校正

递归下降分析程序中的错误校正的一个标准形式叫做应急方式 (panic mode)。这个名称是这样得来的: 在复杂情况下, 错误处理器将可能要试图在大量的记号中找到一个恢复分析的位置 (在最糟的情况下, 它可能甚至会用完所有剩余的记号, 此时只有在错误出现后退出了)。但是, 若在操作时小心一些, 那么在进行错误校正时要比其名称的意味要好一些^①。应急方式还具有一个优点: 它能真正保证在错误校正时, 分析程序不会掉到无穷循环之中。

应急方式的基本机制是为每个递归过程提供一个额外的由同步记号 (synchronizing token) 组成的参数。在分析处理时, 作为同步记号的记号在每个调用发生时被添加到这个集合之中。如若遇到错误, 分析程序就向前扫描 (scan ahead), 并一直丢弃记号直到看到输入中记号的一个同步集合为止, 并从这里恢复分析。在做这种快速扫描时, 通过不生成新的出错信息来 (在某种程度上) 避免错误级联。

在进行该错误校正时需要作出一个重要的判断: 在分析的每步中需添加哪些记号。一般地,

^① 实际上, Wirth [1976] 将应急方式称作 “不应急” 的规则, 这大概是试图要改善它的形象吧。

Follow集合是这样的同步记号中的重要一员。Follow集合也可用来使错误处理器避免跳过开始新的主要结构的重要记号（如语句或表达式）。First集合也很重要，这是因为它们允许递归下降分析程序能在分析的早些时候检测出错误，它对于任何错误校正总是有用的。当编译器“知道”何时不应着急时，应急方式工作得最好，掌握这一点很重要。例如，丢失了诸如分号或逗号，甚至是右括号的标点符号不应总是致使错误处理器耗费记号。当然，必须留意要确保不能发生无穷循环。

我们通过用伪代码在4.1.2节中的递归下降计算器中勾画它的执行过程来讲解应急方式的错误校正（参见程序清单4-1）。除了保持基本相同（但错误再也不能立即退出了）的 *match* 和 *error* 这两个过程之外，还有两个过程，完成早期的先行检查的 *checkinput*，以及应急方式记号耗费者拥有的 *scanto*：

```

procedure scanto ( synchset ) ;
begin
    while not ( token in synchset  { $ } ) do
        getToken ;
    end scanto ;

procedure checkinput ( firstset, followset ) ;
begin
    if not ( token in firstset ) then
        error ;
        scanto ( firstset  followset ) ;
    end if ;
end;

```

这里的\$指的是输入的结尾（EOF）。

这些过程如下所示地用于 *exp* 和 *factor* 过程中（它们现在得到了一个 *synchset* 参数）：

```

procedure exp ( synchset ) ;
begin
    checkinput ( { (, number }, synchset ) ;
    if not ( token in synchset ) then
        term ( synchset ) ;
        while token = + or token = - do
            match ( token ) ;
            term ( synchset ) ;
        end while ;
        checkinput ( synchset, { (, number } ) ;
    end if;
end exp ;

procedure factor ( synchset ) ;
begin
    checkinput ( { (, number }, synchset ) ;
    if not ( token in synchset ) then

```

```

case token of
( : match ( ( );
  exp ( { } ) );
  match ( ) );
number :
  match ( number );
else error ;
end case ;
checkinput ( synchset, { ( , number ) } );
end if ;
end factor ;

```

请读者注意, *checkinput* 在每个过程中都被调用了两次: 一次是核实 First 集合的记号是输入中的下一个记号, 另一次是核实 Follow 集合 (或 *synchset*) 的记号是退出的下一个记号。

应急方式的这种格式将产生合理的错误 (有用的出错信息也可作为参数被添加到 *checkinput* 和 *error* 中)。例如, 输入串 $(2+-3)*4--5$ 将产生两个出错信息 (一个在第1个减号上, 另一个在第2个加号上)。

一般地, 我们注意到在递归调用中向下传送 *synchset*, 同时也添加相应的新同步记号。在 *factor* 的情况中, 当看到一个左括号之后会出现了一个例外: 只有在作为它的 Follow 集合时, *exp* 才与右括号一起被调用 (丢弃 *synchset*)。这是一种伴随应急方式错误校正的典型特殊分析 (这样做了以后, 例如表达式 $(2+*)$ 将不在右括号上生成一个虚假的错误信息。) 我们将该代码行为的分析以及其在 C 中的实现留在练习中。不幸的是, 为了得到最佳的出错信息和错误校正, 在实际中必须检查每个记号测试以看看是否需要做更一般的或更早的测试, 而这样可能会增加错误行为。

4.5.2 在 LL(1) 分析程序中的错误校正

应急方式错误校正也可在 LL(1) 分析程序中以与在递归下降分析中相似的方式实现。由于该算法是非递归的, 所以就要求用一个新栈来保存 *synchset* 参数, 而且在算法生成每个动作之前 (当一个非终结符位于栈顶之时), 算法必须安排一个对 *checkinput* 的调用^①。请读者注意, 在出现最初错误时, 在栈顶部有一个非终结符 *A*, 且当前的输入记号不在 *First(A)* 中 (或若 ϵ 在 *First(A)* 时, 则为 *First(A)*)。记号位于栈的顶部且与当前输入记号不同的情况是不常见的, 这是因为就一般而言, 当在输入中真正地看到记号时, 它们只会被压入到栈中 (表压缩方法可能会和它折衷一点)。我们将把程序清单 4-2 中的分析算法的修改留在练习里。

若不用一个额外的栈, 也可静态地将同步记号的集合与 *checkinput* 所采取的相应动作一起建立到 LL(1) 分析表中。假设有一个位于栈顶部的非终结符 *A* 和一个不在 *First(A)* (或若 ϵ 在 *First(A)* 时, 则为 *First(A)*) 中的输入记号, 那么就有 3 种其他方法:

- 1) 由栈弹出 *A*。
- 2) 看到一个为了它可重新开始分析的记号之后, 成功地从输入中弹出记号。
- 3) 在栈中压入一个新的非终结符。

① 同在递归下降代码中一样, 位于匹配末尾的向 *checkinput* 的调用也能由一个特殊的栈符号以类似于第 4.2.4 节中值计算安排的风格来安排。

若当前输入记号是\$或是在Follow(A)中时,就选择方法1。若当前输入记号不是\$或不在First(A) Follow(A)中,就选择方法2。在特殊情况方法3有时会有用,但却很少是恰当的(后面将简要地讨论另一种方法)。第1个动作是通过记号 pop 在分析表中指出,第2个动作是通过记号 $scan$ 指出(请注意, pop 动作与 ϵ 产生式的归约相等价)。

有了这些惯例之后,LL(1)分析表(表4-4)看起来应同表4-9一样。例如:串 $(2+*)$,使用该表的LL(1)分析程序的行为显示在表4-10中。在这个表中,分析只显示为由第1个错误开始(所以前缀 $(2+$ 早已被成功地匹配了)。这里还是使用缩写词 E 代表 exp ,用 T 代表 $term$ 等等。请注意,在分析成功地再继续之前两个相邻的错误发生了移动。我们可以通过适当的安排来抑制第2个错误的出错信息在第1个错误之后移动,这样分析程序就可在产生任何新的错误信息之前成功地移动一次或多次了。因此,就可避免出错信息级联了。

表4-9 带有错误校正项的LL(1)分析表(表4-4)

$M[N, T]$	(number)	+	-	*	\$
exp	exp	exp	pop	$scan$	$scan$	$scan$	pop
	$term\ exp$	$term\ exp$					
exp	$scan$	$scan$	$exp\ \epsilon$	exp	exp		
				$addop$	$addop$	$scan$	$exp\ \epsilon$
				$term\ exp$	$term\ exp$		
$addop$	pop	pop	$scan$	$addop$	$addop$	$scan$	pop
				+	-		
$term$	$term$	$term$					
	$factor$	$factor$	pop	pop	pop	$scan$	pop
	$term$	$term$					
$term$	$scan$	$scan$	$term\ \epsilon$	$term\ \epsilon$	$term\ \epsilon$	$term$	$term\ \epsilon$
						$mulop$	
						$factor$	
						$term$	
$mulop$	pop	pop	$scan$	$scan$	$scan$	$mulop\ *$	pop
$factor$	$factor$	$factor$	pop	pop	pop	pop	pop
	$(\ exp\)$	number					

表4-10 LL(1)分析程序使用表4-9的移动

分析栈	输入	动作
$\$ E T) E T$	$*)\$$	扫描(错误)
$\$ E T) E T$	$)\$$	弹出(错误)
$\$ E T) E$	$)\$$	$E\ \epsilon$
$\$ E T)$	$)\$$	匹配
$\$ E T$	$\$$	$T\ \epsilon$
$\$ E$	$\$$	$E\ \epsilon$
$\$$	$\$$	接受

4.5.3 在TINY分析程序中的错误校正

正如在附录B中给出的一样, TINY分析程序的错误处理是极为初步的:它只需实现一个格

式非常原始的应急方式恢复，而无需同步集合。`match`过程仅声明错误，说出它发现的是哪个不希望的记号。除此之外，过程`statement`和`factor`在未发现正确选择时就声明错误。若在分析结束时，发现的是一个记号而不是文件的结束，则`parse`过程就声明错误。产生的主要出错信息是“非期望的记号”，它对于用户没有用处。此外，分析程序也不试着避免错误级联。例如，在`write`语句之后添加了一个分号的样本程序

```
...
5:  read  x  ;
6:  if  0  <  x  then
7:    fact  := 1;
8:    repeat
9:      fact  := fact  *  x;
10:     x  := x  - 1
11:   until  x  = 0 ;
12:   write fact ; {<- - BAD SEMICOLON ! }
13: end
14:
```

产生了以下的两个出错信息（当只有一个错误已发生时）：

```
Syntax error at line 13: unexpected token -> reserved word: end
Syntax error at line 14: unexpected token -> EOF
```

当在代码的第2行中删去小于号“<”时，上面的程序就是：

```
...
5:  read  x  ;
6:  if  0  x  then { <- - COMPARISON MISSING HERE! }
7:    fact  := 1;
8:    repeat
9:      fact  := fact  *  x;
10:     x  := x  - 1
11:   until  x  = 0 ;
12:   write fact ;
13: end
14:
```

这样就打印出了4个出错信息：

```
Syntax error at line 6 : unexpected token -> ID, name = x
Syntax error at line 6 : unexpected token -> reserved word: then
Syntax error at line 6 : unexpected token -> reserved word: then
Syntax error at line 7 : unexpected token -> ID, name = fact
```

另一方面，TINY的某些行为是恰当的。例如，一个丢失掉的（不是额外的）分号将只生成一个出错信息，而分析程序则就如同分号一直在那儿一样继续建立正确的语法树，由此就完成了错误校正的一个基本形式。这个行为是由两种代码引起的。第1个是`match`过程并不耗费记号，它引起的行为与插入丢失的记号一样。第2个是已将`stmt_sequence`过程写出来，这样就在一个错误的情况下联结了尽可能多的语法树的匹配。特别地，必须留意在每当找到一个非零指针时都应与属指针相联结（将分析程序过程设计为若发现错误则返回一个零语法树指针）。另外，基于EBNF

```
statement();
while (token==SEMI)
{ match(SEMI);
```

```
statement();
}
```

的`stmt_sequence`体的明显书写方式是用带有一个更为复杂的循环测试代替：

```
statement() ;
while ((token!= ENDFILE) && (token!= END) &&
      (token!= ELSE) && (token!= UNTIL))
{ match(SEMI);
  statement();
}
```

读者会注意到在这个反面的测试中的4个记号包含了`stmt_sequence`的Follow集合。这并不是出了什么错，而是因为测试将或是寻找First集合中的一个记号（同`statement`和`factor`的过程相同）或是寻找不在Follow集合中的一个记号。因为若丢失了一个First符号，分析就会停止，所以后者在错误校正中尤为有效。我们在练习中描述了一个程序的行为，读者可看到若在第1个格式中给出了`stmt_sequence`，则丢失的分号确实将致使跳过程序的剩余部分。

最后，我们还应注意到分析程序的编写方式：当它遇到错误时，它不能进入一个无穷循环（当`match`不损失一个不期望的记号时，读者会担心到这一个问题）。这是因为，在分析过程的任意路径中，最终都会遇到`statement`或`factor`的缺省情况，而两者都会在产生错误信息时消耗掉一个记号。

练习

- 4.1 编写与4.1.2节中`exp`的伪代码相对应的`term`和`factor`的伪代码，以使其可构造出简单算术表达式的语法树。
- 4.2 若有文法 $A \rightarrow (A)A \mid \varepsilon$ ，请写出由递归下降分析该文法的伪代码。
- 4.3 若有文法

```
statement  assign-stmt | call-stmt | other
assign-stmt  identifier := exp
call-stmt  identifier ( exp-list )
```

请写出由递归下降分析该文法的伪代码。

- 4.4 若有文法

```
lexp      number | ( op lexp -seq )
op        + | - | *
lexp -seq lexp -seq lexp | lexp
```

请写出伪代码以通过递归下降来计算一个`exp`的数值（参见第3章的练习3.13）。

- 4.5 利用表4-4识别以下算术表达式的LL(1)分析程序，请写出它们的动作：
 - a. $3+4*5-6$
 - b. $3*(4-5+6)$
 - c. $3-(4+5*6)$
- 4.6 写出利用4.2.2节中第1个表的LL(1)分析程序来识别以下成对括号的串：
 - a. $(())()$
 - b. $(())()$
 - c. $(())()$
- 4.7 若有文法 $A \rightarrow (A)A \mid \varepsilon$ ，
 - a. 为非终结符 A 构造First集合和Follow集合。
 - b. 说明该文法是LL(1)的文法。
- 4.8 考虑文法

```

lexp    atom | list
atom    number | identifier
list    ( lexp -seq )
lexp -seq    lexp -seq lexp | lexp

```

- 消除左递归。
- 为得出的文法的非终结符构造First集合和Follow集合。
- 说明所得的文法是LL(1)文法。
- 为所得的文法构造LL(1)分析表。
- 假设有输入串

(a (b (2)) (c))

请写出相对应的LL(1)分析程序的动作。

4.9 考虑以下的文法 (与练习4.8类似但不同) :

```

lexp    atom | list
atom    number | identifier
list    ( lexp - seq )
lexp -seq    lexp , lexp -seq | lexp

```

- 在该文法中提取左因子。
- 为所得的文法的非终结符构造First集合和Follow集合。
- 说明所得的文法是LL(1)文法。
- 为所得的文法构造LL(1)分析表。
- 假设有输入串

(a , (b , (2)) , (c))

写出相对应的LL(1)分析程序的动作。

4.10 考虑简化了的C声明的以下文法 :

```

declaration    type var-list
type          int | float
var-list       identifier, var-list | identifier

```

- 在该文法中提取左因子。
- 为所得的文法的非终结符构造First集合和Follow集合。
- 说明所得的文法是LL(1)文法。
- 为所得的文法构造LL(1)分析表。
- 假设有输入串

int x,y,z

写出相对应的LL(1)分析程序的动作。

4.11 诸如表4-4中的LL(1)分析表一般总有许多表示错误的空项。在许多情况下,一行中的所有空项可由一单个的缺省项 (default entry) 代替,由此就可将这个表大大地缩小。当非终结符有一个单独的产生式选择或当它有一个 ϵ 产生式时,在非终结符行就可能会出现缺省项。将该想法应用于表4-4,如果可以应用,有可能会出现怎样的问题?

- LL(1)文法会有二义性吗?为什么?
- 二义性文法会是LL(1)文法吗?为什么?

c. 非二义性的文法一定是LL(1)文法吗？为什么？

4.13 说明左递归文法不会是LL(1)文法。

4.14 证明4.3.3节中由其First集合和Follow集合的两个条件所得文法的定理。

4.15 将在记号串的两个集合 S_1 和 S_2 上的算符 定义为： $S_1 \quad S_2 = \{ \text{First}(xy) \mid x \in S_1, y \in S_2 \}$

a. 说明对于任何两个非终结符 A 和 B ，都有 $\text{First}(AB) = \text{First}(A) \quad \text{First}(B)$ 。

b. 说明4.3.3节中定理的两个条件可由某单一条件：若 $A \quad$ 且 $A \quad$ ，则 $(\text{First}(\quad) \quad \text{Follow}(A)) \quad (\text{First}(\quad) \quad \text{Follow}(A))$ 为空来代替。

4.16 若由开始符号到 A 出现的记号串没有推导，则非终结符 A 是无用的。

a. 为该特性给出一个数学形式。

b. 程序设计语言有无可能具有一个无用的符号？请解释原因。

c. 若文法具有一个无用的符号，说明计算同在本章中给出的一样的 First集合和 Follow集合会生成比用于构造真实的LL(1)分析表长得多的产生集合。

4.17 请给出无需改变被识别的语言就可从一个文法中删除无用非终结符（及相结合的产生式）的一个算法（参见前一个练习）。

4.18 若文法没有无用的非终结符，说明4.3.3节中的定理的反命题也是正确的（参见练习4.16）。

4.19 给出例4.15中First集合和Follow集合的详细计算。

4.20 a. 为例4.7提取了左因子的文法中的非终结符构造First集合和Follow集合。

b. 利用(a)部分的答案构造LL(1)分析表。

4.21 若有文法 $A \quad a A a \mid \quad$ ，

a. 说明该文法不是LL(1)文法。

b. 以下伪代码试图写出该文法的递归下降分析程序

```

procedure A ;
begin
    if token = a then
        getToken ;
        A ;
    if token = a then getToken ;
    else error ;
    else if token < > $ then error ;
end A ;

```

说明这个过程不能正确地运行。

c. 可以写出该语言的带回溯（backtracking）递归下降分析程序，但这样却要求使用将一个记号看作参数并将该记号返回到输入记号流前面的 unGettoken过程。它还要求将过程 A 写作返回成功或失败的布尔函数，这样当 A 调用其自身时，它可在耗费另一个记号前先测试一下是否能成功；因此当 $A \quad a A a$ 选择失败，代码还可继续尝试 $A \quad \varepsilon$ 。根据以上所述请重新写出（b）部分的伪代码，并描绘其在串 $aaaa$ 上的运算。

4.22 在程序清单4-8的TINY文法中，并没有将布尔表达式和算术表达式清楚地区分开。例如，以下是一个合乎语法的TINY程序：


```
if 0 then write 1>0 else x := (x<1)+1 end
```

请重新写出TINY文法以便只允许布尔表达式作为if语句或repeat语句的测试，且只允许算术表达式出现在write语句或assign语句中或作为任何算符的操作数。

- 4.23 将布尔算符and、or和not添加到程序清单4-8的TINY文法中。并给其赋予练习3.5中所描述的特性以及比所有算术算符都低的优先权。确保任何表达式都可以是布尔表达式或整型表达式。
- 4.24 对练习4.23中TINY语法的改变使得练习4.22所描述的问题变得更糟了。将练习4.23的答案重新改写以使布尔表达式和算术表达式能够严格地区分开来，并与练习4.22的解法相合并。
- 4.25 如4.5.1节所述：用作简单算术表达式文法的应急方式错误校正仍有一些缺点。其中之一是为算符测试的while循环应在特定的条件下继续运行。例如，表达式(2)(3)就在因子之间丢掉了算符，但是错误处理器在未重新开始分析之前就耗费了第2个因子。请读者重写伪代码以改进这样的行为。
- 4.26 利用表4-9中给出的错误恢复描述在输入(2+-3)*4-+5上的LL(1)分析程序的动作。
- 4.27 重写程序清单4-2中的LL(1)分析算法以使应急方式错误校正保持完整，并描绘其在输入(2+-3)*4-+5中的行为。
- 4.28 a. 描绘在TINY分析程序中的stmt_sequence过程的一个运算，它用于核实构造以下TINY程序的语法树（除了丢失的分号之外）的正确性：

```
x := 2
y := x + 2
```

- b. 可为以下（不正确的）程序构造怎样的语法树：

```
x 2
y := x + 2
```

- c. 假设用编写stmt_sequence过程来代替本章最后一节中的更为简单的代码：

```
statement();
while(token = SEMI)
{ match(SEMI);
  statement();
}
```

利用这种版本的stmt_sequence过程可为a部分和b部分中的程序构造怎样的语法树呢？

编程练习

- 4.29 将以下所述添加到程序清单4-1中的简单整型算术递归下降计算器内（确保它们具有正确的结合性和优先权）：
 - a. 带有符号/的整型除法。
 - b. 带有符号%的整型模。
 - c. 带有符号^的整型求幂（警告：该算符比乘法优先，且是右结合）。
 - d. 带有符号-的一目减（参见练习3.12）。
- 4.30 重写程序清单4-1的递归下降计算器，使其可与浮点数而不单是整型一起计算。
- 4.31 重写程序清单4-1的递归下降计算器，使其可区分浮点数和整型值，而不仅仅是将所有项都作为整型或浮点数来计算（提示：现在的“值”是指带有指示它是个整型还

是浮点的标志的记录)。

- 4.32 a. 重写程序清单 4-1 的递归下降计算器，使其根据 3.3.2 节的声明返回一个语法树。
 b. 写出一个作为参数的函数，它得到由 a 部分的代码生成的语法树，并由移动树来返回计算的值。
- 4.33 为与程序清单 4-1 相似的整型算法写出一个递归下降计算器，但要使用图 4-1 中的文法。
- 4.34 考虑以下的文法：

$$\begin{aligned} \text{lexp} & \quad \text{number} \mid (\text{op lexp -seq}) \\ \text{op} & \quad + \mid - \mid * \\ \text{lexp -seq} & \quad \text{lexp -seq lexp} \mid \text{lexp} \end{aligned}$$

该文法可被看成是表示在类似 LISP 前缀格式中的简单整型算术表达式。例如，表达式 $34 - 3 * 42$ 在该文法中写作 $(- \ 34 \ (* \ 3 \ 42))$ 。

为该文法给出的表达式写出一个递归下降计算器。

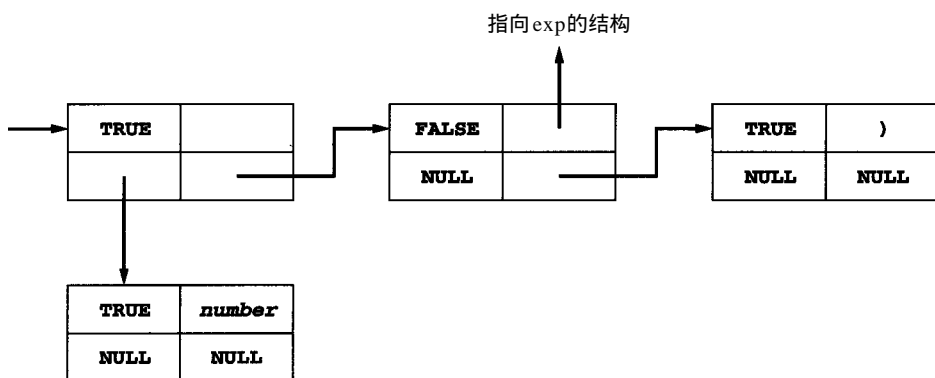
- 4.35 a. 为前一个练习的文法设计一个语法树结构，并为返回语法树的它写出一个递归下降分析程序。
 b. 写出一个作为参数的函数，它得到由 a 部分的代码生成的语法树，并由编号树来返回计算的值。
- 4.36 a. 为练习 3.4 的正则表达式使用（被恰当地消除了二义性）的文法以构造一个读取正则表达式并执行 NFA 的 Thompson 结构（参见第 2 章）的递归下降分析程序。
 b. 编写一个过程，它将 NFA 数据结构看作是由 a 部分的分析程序生成的，且根据子集结构构造等价的 DFA。
 c. 编写一个过程，它将数据结构看作是由 (b) 部分的过程生成的，并根据它所代表的 DFA 在文本文件中寻找最长子串来进行匹配。（你的程序现在变成了 grep 的一个“编译过的”版本了！）
- 4.37 将比较算符 \leq （小于或等于）、 $>$ （大于）、 \geq （大于或等于）以及 \neq （不等于）添加到 TINY 分析程序中（还将要求添加这些记号以及改变扫描程序，但是不应要求语法树有所改变）。
- 4.38 将在练习 4.22 中的文法变化合并到 TINY 分析程序中。
- 4.39 将在练习 4.23 中的文法变化合并到 TINY 分析程序中。
- 4.40 a. 将程序清单 4-1 中的递归下降计算器程序重新改写以完成如在第 4.5.1 中描述的那样的应急方式错误校正。
 b. 给 a 部分中你的错误处理添加有用的出错信息。
- 4.41 TINY 分析程序只能产生极少的出错信息，其中的一个原因是 `match` 过程被限制于在错误出现时只打印当前记号，而不是打印当前记号与所希望的记号，而且从其调用点没有特殊的出错信息被传送到 `match` 过程。将 `match` 过程重新改写以使它在打印当前记号的同时也打印出所希望的记号，并可将出错信息传给 `syntaxError` 过程。这将要求重写 `syntaxError` 过程，以及改变 `match` 调用使其包括恰当的出错信息。
- 4.42 按 4.5.1 节中所述方法，给 TINY 分析程序添加记号的同步集合和应急方式错误校正。
- 4.43 一个可分析 LL(1) 文法规则的任何集合的“一般的”递归下降分析程序可使用基于语法图的（来自 Wirth [1976] 的）数据结构。下面的 C 声明给出了一个适当的数据结构：

```
typedef struct rulerec
{
    struct rulerec *next, *other;
    int isToken;
    union
    {
        Token name ;
        struct rulerec *rule ;
    } attr ;
} Rulerec ;
```

`next`域被用来指出文法规则中的下一个项目，而 `other`域则被用来指出由 | 元符号给出的替换项。因此，用于文法规则

$$\text{factor} \quad (\text{exp}) | \text{number}$$

的数据结构看起来如下所示：



其中记录结构的域如下：

isToken	name/rule
other	next

- 为图4-1中的其余文法规则画出数据结构（提示：这将需要在数据结构内部代表 ϵ 的一个特殊记号）。
- 写出使用这些数据结构识别输入串的一个普通分析过程。
- 写出（从一个文件中或输入中）读取 BNF规则并生成前述的数据结构的分析程序生成器。

注意与参考

用于分析程序构造的递归下降分析自从20世纪60年代以及Algo160报告[Naur, 1963]的BNF规则的介绍中就已有了一个标准方法。若读者希望看到这一方法的较早描述，可参看 Hoare [1962]。回溯递归下降分析程序最近已在诸如 Haskell和Miranda的极为典型的懒惰函数语言中变得流行起来，其中这种递归下降分析的形式被称作组合器分析。读者可在 Peyton Jones 和 Lester [1992]或Hutton [1992]中找到这种方法的描述。在 Wirth [1976]中，将EBNF连同递归下降分析一起使用已十分普通了。

LL(1)分析在20世纪的60年代和70年代早期已得到了广泛的研究。在 Lewis 和 Stearns

[1968]中可看到它的早期描述。在Fischer 和 LeBlanc [1991]中可找到对LL(k)分析的调查，其中还有一个不是SLL(2)的LL(2)文法的示例。Parr、Dietz和Cohen [1992]中有关于LL(k)分析的特殊应用。

当然，除了在本章所学到的两个之外，还有很多自顶向下的分析方法。Graham、Harrison 和 Ruzzo [1980]中有其他的更为普通的方法。

Wirth [1976]和Stirling [1985]中有关于应急方式错误校正的研究。LL(k)分析程序中的错误校正可在Burke 和 Fisher [1987]中找到。读者还可在Fischer and LeBlanc [1991]和Lyon [1974]中学到更复杂的错误修改方法。

本章并未讨论到用于产生自顶向下的分析程序的自动工具，这主要是由于第 5章将讨论最常用的工具——Yacc。但是，好的自顶向下的分析程序生成器还是存在的。Antlr就是其中之一，它是Purdue Compiler Construction Tool Set(PCCTS)的一部分。读者可参看Parr、Dietz和Cohen [1992]中的相关内容。Antlr从EBNF中生成了一个递归下降分析程序。它具有大量有用的特征，这其中就包括了用于构造语法树的内部机制。读者可参看Fischer 和 LeBlanc [1991]中有关称作LLGen的LL(1)分析程序生成器的大致内容。

China-pub.com

下载

第5章 自底向上的分析

本章要点

- 自底向上分析概览
- LR(0)项的有穷自动机及LR(0)分析
- SLR(1)分析
- 一般的LR(1)和LALR(1)分析
- Yacc: LALR(1)分析程序的生成器
- 使用Yacc生成TINY分析程序
- 自底向上分析程序中的错误校正

前一章涉及到了递归下降和预测分析的自顶向下分析的基本算法。在本章中，我们将描述主要的自底向上的分析技术及其相关的构造。如在自顶向下的分析一样，我们将主要学习利用最多一个先行符号的分析算法，此外再谈一下如何扩展这个算法。

与LL(1)分析程序的术语相类似，最普通的自底向上算法称作 LR(1)分析(LR(1) parsing)(L表示由左向右处理输入，R表示生成了最右推导，而数字1则表示使用了先行的一个符号)。自底向上分析的作用还表现在它使得 LR(0)分析(LR(0) parsing)也具有了意义，此时在作出分析决定时没有考虑先行(因为可在先行记号出现在分析栈上之后再检查它，而且倘若是这样发生的，它也就不会被算作先行了，所以这是可能的)。SLR(1)分析(SLR(1) parsing，可简称为LR(1)分析)是对LR(1)分析的改进，它使用了一些先行。LALR(1)分析(LALR(1) parsing，意即先行LR(1)分析)是比SLR(1)分析略微强大且比一般的LR(1)分析简单的方法。

在本章节中将会谈到以上每一个分析方法的必要构造。这将包括 LR(0)的DFA构造以及LR(1)各项：SLR(1)、LR(1)和LALR(1)分析算法的描述，以及相关分析表的构造。我们还将描述Yacc(一个LALR(1)分析程序生成器)的用法，并为TINY语言使用Yacc生成一个分析程序，该TINY语言构造出与在第5章中由递归下降分析程序开发的相同的语法树。

一般而言，自底向上的分析算法的功能比自顶向下的方法强大(例如，左递归在自底向上分析中就不成问题)。但是这些算法所涉及到的构造却更为复杂。因此，需要小心对它们的描述，我们还需要利用文法的十分简单的示例来介绍它们。本章的开头先给出了这样的两个例子，本章还会一直用到这两个例子。此外还会用到一些前章中的例子(整型算术表达式、if语句等等)。但是由于为全TINY语言用手工执行任何的自底向上的分析算法非常复杂，所以我们并不打算完成它。实际上，所有重要的自底向上方法对于手工编码而言都太复杂了，但是对于诸如Yacc的分析程序生成器却很合适。但是了解方法的操作却很重要，这样作为编译程序的编写者就可对分析程序生成器的行为进行正确地分析了。由于分析程序生成器可以用BNF中建议的语言语法来识别可能的问题，所以程序设计语言的设计者还可从这个信息中获益不少。

为了掌握好自底向上的分析算法，读者还需要了解前面讲过的内容，这其中包括了有穷自动机、从一个NFA中构造DFA的子集(第2章的2.3节和2.4节)、上下文无关文法的一般特性、推导和分析树(第3章的3.2节和3.3节)。有时也需要用到Follow集合(第4章的4.3节)。这一章从自底向上的分析的情况开始谈起。

5.1 自底向上分析概览

自底向上的分析程序使用了显式栈来完成分析,这与非递归的自顶向下的分析程序相类似。分析栈包括记号和非终结符,以及一些后面将讨论到的其他信息。自底向上的分析开始时栈是空的,在成功分析的末尾还包括了开始符号。因此,可将自底向上的分析示意为:

```

$                               InputString $
...                               ...
...                               ...
$ StartSymbol                   $      accept

```

其中,分析栈在左边,输入位于正中间,而分析程序的动作则在右边(此时,“接受”是所指出的唯一动作)。

自底向上的分析程序有两种可能的动作(除“接受”之外):

- 1) 将终结符从输入的开头移进到栈的顶部。
- 2) 假设有BNF选择 A ,将栈顶部的串归约为非终结符 A 。

因此自底向上的分析程序有时称作是移进-归约分析程序^①。移进动作是由书写单词shift指出的。归约动作则由书写 $reduce$ 单词给出且指出在归约中所用的BNF选择^②。自底向上的分析程序的另一个特征是:出于后面要讲到的技术原因,总是将文法与一个新的开始符号一同扩充(augmented)。这就意味着若 S 是开始符号,那么就将新的开始符号 S 增加到文法中,同时还添加一个单元产生式到前面的开始符号中:

$$S \rightarrow S$$

下面是两个示例,之后再讨论这两个例子所表现出的自底向上分析的一些特性。

例5.1 考虑以下用于成对括号的扩充文法:

$$\begin{aligned}
 S &\rightarrow S \\
 S &\rightarrow (S)S \mid \varepsilon
 \end{aligned}$$

表5-1给出了使用该文法的串 $()$ 的自底向上的分析。

表5-1 例5.1中文法的自底向上分析程序的分析动作

	分 析 栈	输 入	动 作
1	\$	() \$	移进
2	\$ () \$	用 S 归约
3	\$ (S) \$	移进
4	\$ (S)	\$	用 $S \rightarrow \varepsilon$ 归约
5	\$ (S) S	\$	用 $S \rightarrow (S)S$ 归约
6	\$ S	\$	用 $S \rightarrow S$ 归约
7	\$ S	\$	接受

① 由于相同的原因,自顶向下的分析程序可被称作生成-匹配分析程序,但这并未成为惯例。

② 在归约的情况中,可如同在自顶向下分析中为一个生成动作所做的一样,仅仅由BNF选择自身写出即可,但是惯例却是添加 $reduce$ 。

例5.2 考虑以下基本算术表达式的扩充文法(没有括号,只有一种运算):

$$\begin{aligned} E & \rightarrow E \\ E & \rightarrow E + n \mid n \end{aligned}$$

表5-2是串 $n + n$ 使用这个文法的自底向上的分析。

表5-2 例5.2中文法的自底向上分析程序的分析动作

	分析栈	输入	动作
1	\$	$n + n \$$	移进
2	$\$ n$	$+ n \$$	用 $E \rightarrow n$ 归约
3	$\$ E$	$+ n \$$	移进
4	$\$ E +$	$n \$$	移进
5	$\$ E + n$	$\$$	用 $E \rightarrow E + n$ 归约
6	$\$ E$	$\$$	用 $E \rightarrow E$ 归约
7	$\$ E$	$\$$	接受

在使用先行上,自底向上的分析程序比自顶向下的分析程序的困难要小。实际上,自底向上的分析程序可将输入符号移进到栈里直到它判断出要执行的是何种动作为止(假设判断一个动作可以不要求将符号移进到输入中)。但是为了判断要完成的动作,自底向上的分析程序需要在栈内看得更深,而不仅仅是顶部。例如,在表5-1中的第5行, S 位于栈的顶部,且分析程序用产生式 $S \rightarrow (S) S$ 归约,而第6行在栈的顶部也有 S ,但是分析程序却用 $S \rightarrow S$ 归约。为了能够了解 $S \rightarrow (S) S$ 在第5步中是一个有效的归约,我们必须知道实际上栈在该点包含了串 $(S) S$ 。因此,自底向上的分析要求任意的“栈先行”。由于分析程序本身建立了栈且可使恰当的信息成为可用的,所以这几乎同输入先行一样重要。此时所用的机制是“项”的确定性的有穷自动机,下一节将会讲到它。

当然,仅仅了解栈的内容并不足以可唯一地判断出移进-归约分析的下一步,还需将输入中的记号作为一个先行来考虑。例如在表5-2的第3行, E 在栈中,且发生了一个移进;但在第6行中, E 又在栈中,但却用了 $E \rightarrow E$ 归约。两者的区别在于:在第3行中,输入的下一个记号是“+”;但在第6行中,下一个输入记号却是 $\$$ 。因此,任何执行那个分析的算法必须使用下一个输入记号(先行)来判断出适当的动作。不同的移进-归约分析方法以不同的方式使用先行,而这将导致具有不同能力和复杂性的分析程序。在看到单个算法之前,我们首先做一些普通的观察来看看自底向上分析的直接步骤是如何表现特征的。

首先,我们再次注意到移进-归约分析程序描绘出输入串的最右推导,但推导步骤的顺序却是颠倒的。在表5-1中,共有4个归约,它们与最右推导的4个步骤对应的顺序相反:

$$S \rightarrow S \rightarrow (S) \rightarrow S(S) \rightarrow ()$$

在表5-2中,相对应的推导是

$$E \rightarrow E \rightarrow E + n \rightarrow n + n$$

我们将这样的推导中的终结符和非终结符的每个中间串都称作右句型(right sentential form)。在移进-归约分析中,每个这样的句型都被分析栈和输入分隔开。例如,发生在前面推导中的第3步的右句子格式 $E + n$ 出现在表5-2的第3、第4和第5步。如果通过符号 \parallel 指出每一时刻栈的顶部位于何处(即:当在栈和输入之间发生了分隔),则表5-2的第3步就由 $E \parallel + n$ 给出,而其第4步则由 $E + \parallel n$ 给出。在每一种情况下,分析栈的符号序列都被称作右句型的可行前缀

(viable prefix)。因此， E 、 $E +$ 和 $E + n$ 都是右句型 $E + n$ 的可行前缀，但右句子格式 $n + n$ 却使 ε 和 n 作为它的可行前缀(表5-2的第1步和第2步)。请注意， $n +$ 不是 $n + n$ 的可行前缀。

移进-归约分析程序将终结符从输入移进到栈直到它能执行一个归约以得到下一个右句子格式。它发生在位于栈顶部的符号串匹配用于下一个归约的产生式的右边。这个串、它在右句子格式中发生的位置以及用来归约它的产生式被称作右句型的句柄(handle)^①。例如，在右句子格式 $n + n$ 中，它的句柄是由最左边的单个记号 n 与用来归约它以产生新的右句型 $E + n$ 的产生式 $E \rightarrow n$ 组成的串。这个新句型的句柄是整个串 $E + n$ (一个可行的前缀)以及产生式 $E \rightarrow E + n$ 。有时由于表示法上的弊端，我们要用串本身来作为句柄。

判断分析中的下一个句柄是移进-归约分析程序的主要任务。请注意，句柄串总是为它的产生式(在下一步的归约中使用的产生式)构成一个完整的右部，而且当归约发生时，句柄串的最右边的位置将与栈的顶部相对应。所以对于移进-归约分析程序将要基于产生式右边的位置来判断它的动作这一点而言，看起来有些似是而非了。当这些位置到达产生式的右边末端时，这个产生式就有可能是一个归约，而且句柄还有可能位于栈的顶部。但为了成为句柄，串位于栈的顶部来匹配产生式的右边并不够。实际上，若 ε 产生式可用于归约的话(如在例5.1中)，那么它的右边(空串)总是位于栈的顶部。归约仅发生在结果串实际为一个右句型时。例如，在表5-1的第3步中，可完成用 $S \rightarrow \varepsilon$ 归约，但是得到的串 $(S S)$ 并不是右句子格式，所以 ε 在句子格式 (S) 中的这个位置也不是句柄。

5.2 LR(0)项的有穷自动机与LR(0)分析

5.2.1 LR(0)项

上下文无关文法的 **LR(0)项**(LR(0) item)(或简称为项(item))是在其右边带有区分位置的产生式选择。我们可用一个句点(当然它就变成了元符号，而不会与真正的记号相混淆)来指出这个区分的位置。所以若 $A \rightarrow \dots$ 是产生式选择，且若 x 和 y 是符号的任何两个串(包括空串 ε)，且存在着 $xy = \dots$ ，那么 $A \rightarrow \dots x$ 就是LR(0)项。之所以称作LR(0)项是由于它们不包括先行的显式引用。

例5.3 考虑例5.1的文法：

$$\begin{aligned} S &\rightarrow S \\ S &\rightarrow (S)S|\varepsilon \end{aligned}$$

这个文法存在着3个产生式选择和8个项目：

$$\begin{aligned} S &\rightarrow \cdot S \\ S &\rightarrow S \cdot \\ S &\rightarrow \cdot (S)S \\ S &\rightarrow (\cdot S)S \\ S &\rightarrow (S \cdot)S \\ S &\rightarrow (S) \cdot S \\ S &\rightarrow (S)S \cdot \end{aligned}$$

① 如果文法有二义性，那么由此就会存在多于一个的推导，则在右句型中就会有多于一个的句柄。如果文法没有二义性，则句柄就是唯一的。

$$S \quad .$$

例5.4 例5.2的文法存在着以下的8个项目：

$$\begin{aligned} E & \quad .E \\ E & \quad E. \\ E & \quad .E + n \\ E & \quad E. + n \\ E & \quad E + .n \\ E & \quad E + n. \\ E & \quad .n \\ E & \quad n. \end{aligned}$$

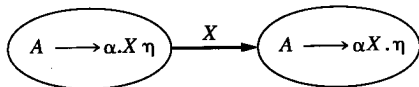
项目概念的思想就是指项目记录了特定文法规则右边识别中的中间步骤。特别地，项目 $A \quad .$ 是由文法规则选择 A 构成（其中 $=$ ），这一点意味着早已看到了 A ，且可能从下一个输入记号中获取 $.$ 。从分析栈的观点来看，这就意味着 A 必须出现在栈的顶部。项目 $A \quad .$ 意味着将要利用文法规则选择 A 识别 A （将这样的项目称作初始项(initial item)）。项目 $A \quad .$ 意味着现在位于分析栈的顶部，而且若 A 在下一个归约中使用的话，它有可能就是句柄（将这样的项目称作完整项 (complete item)）。

5.2.2 项目的有穷自动机

LR(0)项可作为一个保持有关分析栈和移进-归约分析过程的信息的有穷自动机的状态来使用。这将从作为非确定性的有穷自动机开始。从这个 LR(0)项的NFA中可利用第2章的子集构造来构建出LR(0)项集合的DFA。正如将要看到的一样，直接构造LR(0)项集合的DFA也是很简单的。

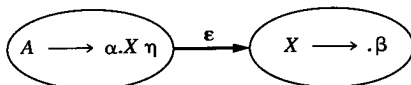
LR(0)项的NFA的转换是什么呢？若有项目 $A \quad .$ ，且假设 X 以符号 X 开始，其中 X 可以是记号或非终结符，所以该项目就可写作 $A \quad .X$ 。那么在符号 X 上就有一个从由该项目代表的状态到由项目 $A \quad X.$ 代表的状态的转换。把它写在一般格式中，就是：

若 X 是一个记号，那么该转换与 X 的一个在分析中从输入到栈顶部的移进相对应。另一方面，



若 X 是一个非终结符，因为 X 永远不会作为输入符号出现，所以该转换的解释就复杂了。实际上，这样的转换仍与在分析时将 X 压入到栈中相对应，但是它只发生在由产生式 $X \rightarrow \beta$ 形成的归约时。由于这样的归约前面必须有一个 X 的识别，而且由初始项 $X \rightarrow .\beta$ 给出的状态代表了这个处理的开始（句点指出将要识别一个 X ），则对于每个项目 $A \quad .X$ ，必须为 X 的每个产生式 $X \rightarrow \beta$ 添加一个 ϵ 产生式，

它指示通过识别它的产生式的右边的任意匹配来产生 X 。



这两种情况只表示 LR(0)项的NFA中的转换，它并未讨论到 NFA的初始状态和接受状态。

例5.7 考虑图5-1的NFA。相关的DFA的开始状态是由项目 $S \rightarrow \cdot S$ 组成的集合的 ϵ -闭包，而这又是3个项目的集合 $\{S \rightarrow \cdot S, S \rightarrow \cdot (S), S \rightarrow \cdot)\}$ 。由于 S 有一个从 $S \rightarrow \cdot S$ 到 $S \rightarrow S \cdot$ 的转换，所以也就存在着一个从开始状态到DFA状态 $\{S \rightarrow S \cdot\}$ 的对应转换(不存在从 $S \rightarrow \cdot S$ 到任何其他项目的 ϵ -

转换)。在(上也有一个从开始状态到DFA状态的转换 $\{S \rightarrow (.S)S, S \rightarrow (.S)S, S \rightarrow (.S)S\}$ 的 ϵ 闭包)。DFA状态 $\{S \rightarrow (.S)S, S \rightarrow (.S)S, S \rightarrow (.S)S\}$ 在(上有到其自身的转换,在S上也有到 $\{S \rightarrow (S.)S, S \rightarrow (S.)S, S \rightarrow (S.)S\}$ 的转换。这个状态在(上有到状态 $\{S \rightarrow (S.)S, S \rightarrow (S.)S, S \rightarrow (S.)S\}$ 的转换。最后,这一最终状态在(上有到前面所构造的状态 $\{S \rightarrow (.S)S, S \rightarrow (.S)S, S \rightarrow (.S)S\}$ 的转换。图5-3是完整的DFA,为了便于引用还给各个状态编了号(按照惯例,状态0是开始状态)。

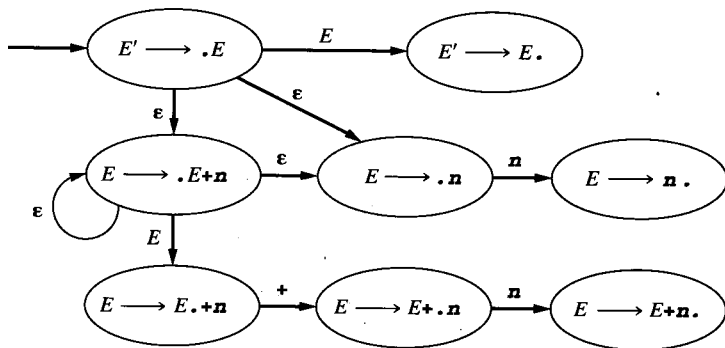


图5-2 例5.6中文法的LR(0)项的NFA

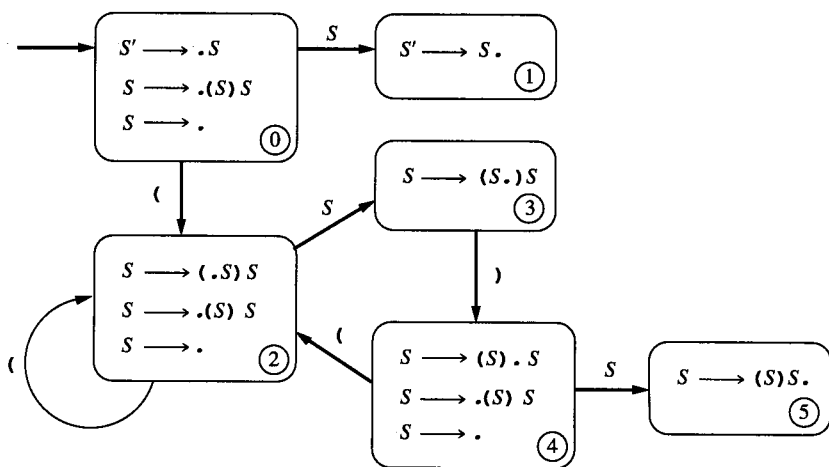


图5-3 与图5-1的NFA相对应的项目集的DFA

例5.8 考虑图5-2的NFA。相关DFA的开始状态由3个项目的集合 $\{E \rightarrow .E, E \rightarrow .E + n, E \rightarrow .n\}$ 组成。在E上有一个从项目 $E \rightarrow .E$ 到项目 $E \rightarrow E.$ 的转换,但在E上还有一个从项目 $E \rightarrow .E + n$ 到项目 $E \rightarrow E. + n$ 的转换。因此,在E上就有一个从DFA的开始状态到集合 $\{E \rightarrow E., E \rightarrow E. + n\}$ 的闭包的转换。由于没有任何一个来自这些项目的 ϵ 转换,所以这个集合就是它自身的 ϵ 闭包,且构成了一个完整的DFA状态。来自开始状态还有另一个转换,它与符号n上的从 $E \rightarrow .n$ 到 $E \rightarrow n.$ 的转换相对应。因为没有来自项目 $E \rightarrow .n$ 的 ϵ 转换,所以这个项目是它自身的 ϵ 闭包,且构成了一个完整的DFA状态 $\{E \rightarrow n.\}$ 。由于没有来自这个状态的转换,所以被计算的唯一转换仍来自于集合 $\{E \rightarrow E., E \rightarrow E. + n\}$ 。从该集合开始只有一个转换,它与符号+上的从项目 $E \rightarrow E. + n$ 到项目 $E \rightarrow E + n.$ 的转换相对应。项目 $E \rightarrow E + n.$ 也没有 ϵ 转换,所以就在DFA中构造了一个单独的集合。最后,n上有一个从集合 $\{E \rightarrow E + n.\}$ 到集合 $\{E \rightarrow E + n.\}$ 的转换。图5-4

中是整个DFA。

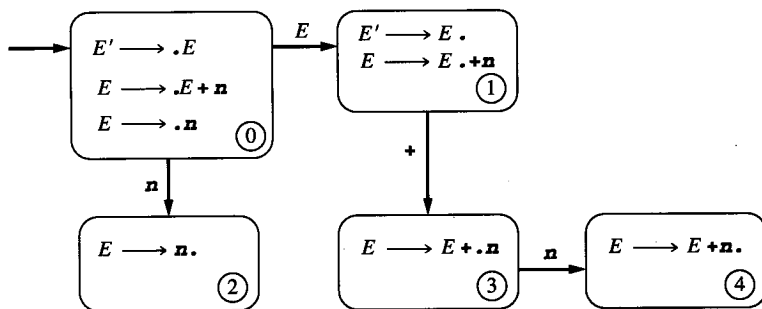


图5-4 与图5-2中的NFA相对应的项目集的DFA

在LR(0)项集的DFA中，有时需要区分在 ϵ 闭包步骤中添加到状态中的项目与引起状态作为非 ϵ -转换的目标的那些项目。前者被称作闭包项 (closure item)，后者被称作核心项 (kernel item)。在图5-4的状态0中， $E' \rightarrow \cdot E$ 是(唯一的)核心项，而 $E \rightarrow \cdot E+n$ 和 $E \rightarrow \cdot n$ 则是闭包项。在图5-3的状态2中， $S \rightarrow (\cdot S)S$ 是核心项，而 $S \rightarrow (\cdot S)S$ 和 $S \rightarrow \cdot$ 是闭包项。根据项目的NFA的 ϵ 转换的定义，所有的闭包项都是初始项。

区分核心项与闭包项的重要性在于：若有一个文法，核心项唯一地判断出状态以及它的转换，那么只需指出核心项就可以完整地表示出项目集的DFA来。因此，构造DFA的分析程序生成器只报告核心项(例如对于Yacc而言，这也是正确的)。

若项目集的DFA是直接运算的，这样就比首先计算项目的NFA之后再应用子集构造要更简化一些。从项目的集合确实可很容易地立即判断出什么是 ϵ -转换以及初始项指向哪里。因此，如Yacc的分析程序生成器总是从文法直接计算DFA，本章后面的内容也是这样做的。

5.2.3 LR(0)分析算法

现在开始讲述LR(0)分析算法。由于该算法取决于要了解项目集的DFA的当前状态，所以须修改分析栈以使不但能存储符号还能存储状态数。这是通过在压入一个符号之后再新的状态数压入到分析栈中完成的。实际上，状态本身就包含了有关符号的所有信息，所以可完全将符号省掉而只在分析栈中保存状态数。但是为了方便和清晰起见，我们仍将在栈中保留了符号。

为了能开始分析，我们将底标记\$和开始状态0压入到栈中，所以分析在开始时的状况表示为：

分析栈	输入
\$ 0	InputString \$

现在假设下一步是将记号 n 移进到栈中并进入到状态2(当DFA如在图5-4中所示一样，且 n 是输入中的下一个记号时，结果就是这样的)。表示如下：

分析栈	输入
\$ 0 n 2	InputString \$ 的剩余部分

LR(0)分析算法根据当前的DFA状态选择一个动作，这个状态总是出现在栈的顶部。

定义：LR(0)分析算法(LR(0) parsing algorithm)。令 s 为当前的状态(位于分析栈的顶部)。则动作定义如下：

1. 若状态 s 包含了格式 $A \rightarrow X$ 的任何项目，其中 X 是一个终结符，则动作就是将当前的输入记号移进到栈中。若这个记号是 X ，且状态 s 包含了项目 $A \rightarrow X$ ，则压入到栈中的新状态就是包含了项目 $A \rightarrow X$ 的状态。若由于位于刚才所描述的格式的状态 s 中的某个项目，这个记号不是 X ，则声明一个错误。
2. 若状态 s 包含了任何完整的项目(格式 $A \rightarrow \cdot$ 的一个项目)，则动作是用规则 A 归约。假设输入为空，用规则 $S \rightarrow S$ 归约(其中 S 是开始状态)与接受相等价；若输入不为空，则出现错误。在所有的其他情况中，新状态的计算如下：将串及其的所有对应状态从分析栈中删去(根据DFA的构造方式，串必须位于栈的顶部)。相应地，在DFA中返回到由开始构造的状态中(这须是由串的删除所揭示的状态)。由于构造DFA，这个状态就还须包含格式 $B \rightarrow A$ 的一个项目。将 A 压入到栈中，并压入包含了项目 $B \rightarrow A$ 的状态(作为新状态)。(请注意，由于正将 A 压入到栈中，所以这与在DFA中跟随 A 的转换相对应(这实际上是合理的)。

若以上的规则都是无歧义的，则文法就是LR(0)文法(LR(0) grammar)。这就意味着若一个状态包含了完整项目 $A \rightarrow \cdot$ ，那么它就不能再包含其他项目了。实际上，若这样的状态还包含了一个“移进的”项目 $A \rightarrow X$ (X 是一个终结符)，就会出现一个到底是执行动作(1)还是执行动作(2)的二义性。这种情况称作移进-归约冲突(shift-reduce conflict)。类似地，如果这样的状态包含了另一个完整项目 $B \rightarrow \cdot$ ，那么也会出现一个关于为该归约使用哪个产生式(A 或 B)二义性。这种情况称作归约-归约冲突(reduce-reduce conflict)。所以，当仅当每个状态都是移进状态(仅包含了“移进”项目的状态)或包含了单个完整项目的归约时，该文法才是LR(0)。

我们注意到上面例子中所用到的两个文法都不是LR(0)文法。在图5-3中，状态0、状态2和状态4都包括了对于LR(0)分析算法的移进-归约冲突；在图5-4的DFA中，状态1包含了一个移进-归约冲突。由于几乎所有“真正的”文法都不是LR(0)，所以这并不奇怪。但下面将会给出一个文法是LR(0)的示例。

例5.9 考虑文法

$$A \rightarrow (A) \mid a$$

扩充的文法具有如图5-5所示的项目集合的DFA，而这就是LR(0)。为了看清LR(0)分析算法是如何工作的，可考虑一下串 $((a))$ 。该串的分析是根据表5-3各步骤所给出的LR(0)分析算法进行。分析开始于状态0，因为这个状态是一个移进状态，所以将第1个记号(移进到栈中。接着，由于DFA指出了在符号 $($ 上从状态0到状态3的转换，所以将状态3压入到栈中。状态3也是一个移进状态，所以下一个 $($ 也被移进到栈中，而且在 $($ 上的转换返回到状态3。移进再一次将 a 放入到栈中，而且 a 上的转换由状态3进入到状态2。现在位于表5-3的第4步，而且已到达了第1个归约状态。这里的状态2和符号 a 都是由栈弹出的，并回到处理中的状态3。接着，将 A 压入到栈中，且得到由状态3到状态4的 A 转换。状态4是一个移进状态，所以 $)$ 被移进到栈中，且 $)$ 上的转换使分析转到状态5。这里发生了一个由规则 $A \rightarrow (A)$ 进行的归约，它从栈中弹出状态5、状态4、状态3以及符号 $)$ 、 A 和 $($ 。现在的分析位于状态3中，而且 A 和

状态4又被压入到栈中。)再一次被移进到栈中,且压入状态5。由A (A)进行的另一个归约从栈中(向后地)删除了串(3 A 4) 5,而将分析留在状态0中。现在A已被压入且也得到了由状态0到状态1的A转换。状态1是接受状态。由于输入现在是空的,则分析算法承认它。

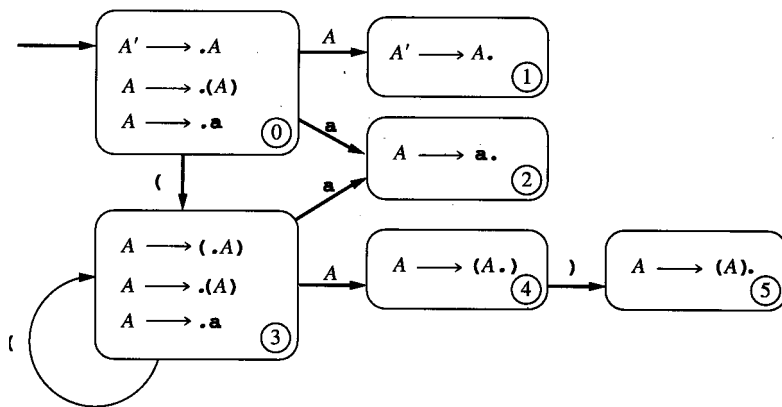


图5-5 例5.9的项目集的DFA

表5-3 例5.9的分析动作

	分析栈	输入	动作
1	\$ 0	((a)) \$	移进
2	\$ 0 (3	(a)) \$	移进
3	\$ 0 (3 (3	a)) \$	移进
4	\$ 0 (3 (3 a 2)) \$	用A a归约
5	\$ 0 (3 (3 A 4)) \$	移进
6	\$ 0 (3 (3 A 4) 5) \$	用A (A) 归约
7	\$ 0 (3 A 4) \$	移进
8	\$ 0 (3 A 3) 5	\$	用A (A) 归约
9	\$ 0 A 1	\$	接受

也可将项目集的 DFA 以及由 LR(0) 分析算法指定的动作合并到分析表中, 所以 LR(0) 分析就变成一个表驱动的分析方法了。其典型结构是: 表的每行用 DFA 的状态标出, 而每一个列也如下标出。由于 LR(0) 分析状态是“移进的”状态或“归约的”状态, 所以专门利用一列来为每个状态指出它。若是“归约的”状态, 就用另一个列来指出在归约中所使用的文法规则选择。若是移进的状态, 将被移进的符号会判断出下一个状态 (通过 DFA), 所以每个记号都会有一个列, 而这些记号的各项都是有关该记号移进后将移到的新状态。由于不能在输入中真正地看到它们, 所以尽管分析程序的行为好像是它们被移进了, 非终结符上的转换 (在归约时被压入) 仍代表着一个特殊情况。因此, 在“移进的”状态中还需要为每个非终结符有一个列, 而且按照惯例, 这些列被列入到称为 goto 部分的表的一个单独部分中。

表5-4就是这样的分析表的一个例子, 它是例 5.9 中文法的表格。读者可以证实这个表将引出如在表5-3中给出的示例的分析动作。

表5-4 例5.9中的文法的分析表

状 态	动 作	规 则	输 入			Goto
			(a)	
0	移进		3	2		1
1	归约	$A \rightarrow A$				
2	归约	$A \rightarrow a$				
3	移进		3	2		4
4	移进				5	
5	归约	$A \rightarrow (A)$				

在这样的分析表中的空白项表示的是错误。当必须要进行错误恢复时，则需要准确地指出分析程序要为每个这些空白项采取什么动作。后面一节将讨论这个问题。

5.3 SLR(1)分析

5.3.1 SLR(1)分析算法

简单LR(1)分析，或SLR(1)分析，也如上一节中一样使用了LR(0)项目集合的DFA。但是，通过使用输入串中下一个记号来指导它的动作，它大大地提高了LR(0)分析的能力。它通过两种方法做到这一点。首先，它在一个移进之前先考虑输入记号以确保存在着一个恰当的DFA。其次，它使用如4.3节所构造的非终结符的Follow集合来决定是否应执行一个归约。令人吃惊的是，先行的这个简单应用的能力强大得足以分析几乎所有的一般语言构造。

定义：SLR(1)分析算法(SLR(1) parsing algorithm)。令 s 为当前状态(位于分析栈的顶部)。则动作可定义如下：

1. 若状态 s 包含了格式 $A \rightarrow \cdot X$ 的任意项目，其中 X 是一个终结符，且 X 是输入串中的下一个记号，则动作将当前的输入记号移进到栈中，且被压入到栈中的新状态是包含了项目 $A \rightarrow X \cdot$ 的状态。
2. 若状态 s 包含了完整项目 $A \rightarrow \cdot$ ，则输入串中的下一个记号是在 $\text{Follow}(A)$ 中，所以动作是用规则 $A \rightarrow \cdot$ 归约。用规则 $S \rightarrow S$ 归约与接受等价，其中 S 是开始状态；只有当下一个输入记号是 $\$$ 时，这才会发生 \odot 。在所有的其他情况中，新状态都是如下计算的：删除串 \cdot 和所有它的来自分析栈中的对应状态。相对应地，DFA回到开始构造的状态。通过构造，这个状态必须包括格式 $B \rightarrow \cdot A$ 的一个项目。将 A 压入到栈中，并将包含了项目 $B \rightarrow A \cdot$ 的状态压入。
3. 若下一个输入记号都不是上面两种情况所提到的，则声明一个错误。

若上述的SLR(1)分析规则并不导致二义性，则文法为**SLR(1)文法**(SLR(1) grammar)。特别地，当且仅当对于任何状态 s ，以下的两个条件：

- 1) 对于在 s 中的任何项目 $A \rightarrow \cdot X$ ，当 X 是一个终结符，且 X 在 $\text{Follow}(B)$ 中时， s 中没有完整的项目 $B \rightarrow \cdot$ 。
- 2) 对于在 s 中的任何两个完整项目 $A \rightarrow \cdot$ 和 $B \rightarrow \cdot$ ， $\text{Follow}(A) \cap \text{Follow}(B)$ 为空。

\odot 实际上，任何文法扩充的开始状态 S 的Follow集合总是只由 $\$$ 组成，这是因为 S 只出现在文法规则 $S \rightarrow S$ 中。

均满足时，文法为SLR(1)。

若第1个条件不满足，就表示这是一个移进-归约冲突(shift-reduce conflict)。若第2个条件不满足，就表示这是一个归约-归约冲突(reduce-reduce conflict)。

这两个条件同前一章中所述的LL(1)分析的两个条件在本质上是类似的。但是如同使用所有的移进-归约分析方法一样，可将决定使用哪个文法规则推迟到最后，同时还可考虑一个更强大的分析程序。

SLR(1)分析的分析表也可以用与前一节所述的LR(0)分析的分析表的类似方式构造。两者的差别如下：由于状态在SLR(1)分析程序中可以具有移进和归约(取决于先行)，输入部分中的每项现在必须要有一个“移进”或“归约”的标签，而且文法规则选择也必须被放在标有“归约”的项中。这还使得动作和规则列成为多余。由于输入结束符号\$也可成为一个合法的先行，所以必须为这个符号在输入部分建立一个新的列。我们将SLR(1)分析表的构造放在SLR(1)分析的第1个示例中。

例5.10 考虑例5.8中的文法，它的项目集合的DFA已列在了图5-4中。正如前面所述的，这个文法不是LR(0)，而是SLR(1)。非终结符的Follow集合是 $\text{Follow}(E) = \{\$, +\}$ 和 $\text{Follow}(E) = \{\$, +\}$ 。表5-5是SLR(1)分析表。在表中，移进由表项中的字母s指出，归约由字母r指出。因此，在输入+的状态1中，指出了一个移进，以及一个到状态3的转换。另一方面，在输入+的状态2中，指出了利用产生式 $E \rightarrow n$ 归约。在输入\$的状态1中还用动作“接受”代替了 $r(E \rightarrow E)$ 。

表5-5 例5.10的SLR(1)分析表

状 态	输 入			Goto
	n	+	\$	E
0	s2			1
1		s3	接受	
2		r($E \rightarrow n$)	r($E \rightarrow n$)	
3	s4			
4		r($E \rightarrow E + n$)	r($E \rightarrow E + n$)	

这个示例的最后是串 $n + n + n$ 的分析。表5-6是它的分析步骤。该图的步骤1以输入记号n的状态0开始，接着分析表指出动作“s2”，即：将记号移进到栈中并进入到状态2。在表5-6中，将它与阶段“shift 2”一起指出来。在该图的步骤2中，分析程序是在状态2中且带有输入记号+，表还指出了用规则 $E \rightarrow n$ 归约。此时，从栈中弹出状态2和记号n。使状态0暴露出来。将符号E压入且将E的Goto从状态0带到状态1。第3步中的分析程序是带有输入记号+的状态1，且表还指出了移进以及指向状态3的转换。在输入n的状态3中，表也指出了移进和到状态4的转换。在输入+的状态4中，表指出用规则 $E \rightarrow E + n$ 归约。这个归约是由将串 $E + n$ 和与它相结合的来自栈的状态弹出来完成的，并再一次暴露状态0，将E压入并将Goto带到状态1中。分析的其他步骤是类似的。

表5-6 例5.10的分析动作

	分 析 栈	输 入	动 作
1	\$ 0	$n + n + n \$$	移进2

(续)

	分析栈	输入	动作
2	\$ 0 n 2	+ n + n \$	用E n归约
3	\$ 0 E 1	+ n + n \$	移进3
4	\$ 0 E 1 + 3	n + n \$	移进4
5	\$ 0 E 1 + 3 n 4	+ n \$	用E E + n 归约
6	\$ 0 E 1	+ n \$	移进3
7	\$ 0 E 1 + 3	n \$	移进4
8	\$ 0 E 1 + 3 n 4	\$	用E E + n 归约
9	\$ 0 E 1	\$	接受

例5.11 考虑成对括号的文法，图 5-3已给出了它的LR(0)项的DFA。一个直接的运算生成了Follow (S) = {\$}和Follow (S) = {\$,)}。表5-7给出了它的SLR(1)分析表。请读者注意，非LR(0)状态0、2和4是如何通过ε- 产生式S ε 具有移进和归约动作的。表 5-8给出了SLR(1)分析算法用来分析串() ()的步骤。请注意，栈如何继续扩展到最终的归约。这是自底向上的分析程序在诸如S (S) S的右递归规则中的一个特征。因此，右递归可引起栈的溢出，所以若可能的话应尽量避免。

表5-7 例5.11的SLR(1)分析表

状态	输入			Goto
	()	\$	S
0	s2	r (S ε)	r (S ε)	1
1			接受	
2	s2	r (S ε)	r (S ε)	3
3		s4		
4	s2	r (S ε)	r (S ε)	5
5		r (S (S) S)	r (S (S) S)	

表5-8 例5.11的分析动作

	分析栈	输入	动作
1	\$ 0	() () \$	移进2
2	\$ 0 (2) () \$	用S ε 归约
3	\$ 0 (2 S 3	() \$	移进4
4	\$ 0 (2 S 3) 4	() \$	移进2
5	\$ 0 (2 S 3) 4 (2) \$	用S ε 归约
6	\$ 0 (2 S 3) 4 (2 S 3) \$	移进4
7	\$ 0 (2 S 3) 4 (2 S 3) 4	\$	用S ε 归约
8	\$ 0 (2 S 3) 4 (2 S 3) 4 S 5	\$	用S (S) S归约
9	\$ 0 (2 S 3) 4 S 5	\$	用S (S) S归约
10	\$ 0 S 1	\$	接受

5.3.2 用于分析冲突的消除二义性规则

SLR(1)分析中以及所有的移进-归约分析方法中的分析冲突都可分为两类：移进-归约冲突和归约-归约冲突。在移进-归约冲突中，有一个自然的消除二义性规则，它总是选取移进而不是归约，因此大多数的移进-归约分析程序通过选择移进来取代归约，也就自动地解决了移进-归约冲突。但是归约-归约冲突就要复杂一些了：这样的冲突通常（但并不是总是）指出文法设计中的一个错误（后面将给出这样冲突的示例）。在移进-归约冲突中选取移进取代归约自动地合并了用于在if语句中的悬挂else二义性的最近嵌套规则，例5.12就是这样的一个例子。这是为什么在程序设计语言的文法中仍保留有二义性的一个原因。

例5.12 考虑在前面章节中所用到的简化了的if语句的文法(例如可参见第3章的3.4.3节)：

```

statement  if-stmt | other
if-stmt   if ( exp ) statement
          | if ( exp ) statement else statement
exp       0 | 1

```

由于这是一个有二义性的文法，所以在任何的分析算法中都会出现分析冲突。为了在SLR(1)分析程序中看到这一点，我们将文法再简化一些使得项目集合的DFA的构造更易处理。甚至还可将测试表达式全部省略，那么文法就如下所示(它仍包含了悬挂else二义性)：

```

S  I | other
I  if S | if S else S

```

图5-6是项目集合的DFA。构造SLR(1)分析动作需要S和I的Follow集合，它们是

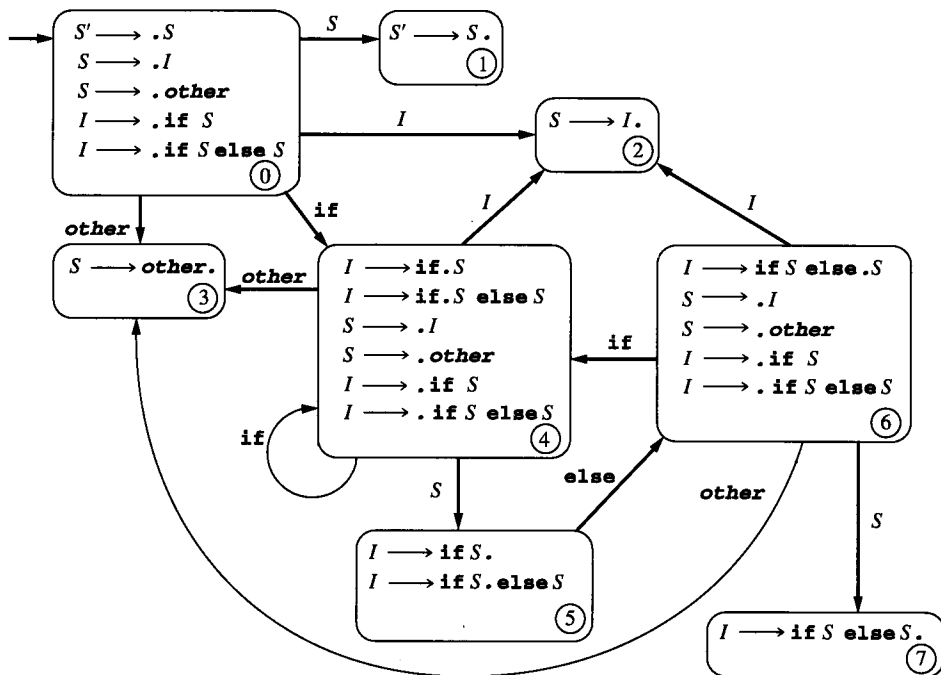


图5-6 例5.12中LR(0)项目集合的DFA

$$\text{Follow}(S) = \text{Follow}(I) = \{\$, \text{else}\}$$

现在就可以看到由悬挂else问题引起的分析冲突了。当发生在DFA的状态5中时，其中的完整项目 $I \text{ if } S$ 指出规则 $I \text{ if } S$ 的归约将发生在输入else和\$中，但项目 $I \text{ if } S \text{ else } S$ 却指出输入记号的一个移进将发生在else上。因此悬挂else将导致在SLR(1)分析表的移进-归约冲突。很明显，用移进取代归约的消除二义性的规则可以消除这个冲突，并会根据最近嵌套规则作出分析(若用归约取代移进，就没有办法在DFA中输入状态6或状态7，这将导致虚假的分析错误)。

表5-9是由该文法引出的SLR(1)分析表。在该表中为归约动作中的文法规则选择使用了编号，用它来代替写出规则本身。编号如下：

- (1) $S \quad I$
- (2) $S \quad \text{other}$
- (3) $I \quad \text{if } S$
- (4) $I \quad \text{if } S \text{ else } S$

请注意，无需为扩充产生式 $S \quad S$ 编号，这是由于用该规则实现的归约与接受相对应，且在表中已被写作“接受”了。

读者应注意到在归约项中使用的产生式编号容易引起与在移进和Goto项中所用到的编号混淆。例如，在表5-9的表的状态5中，输入else下的项目是s6，它指出一个移进以及到状态6的转换，但在输入\$下的项目却是r3，它指出用产生式编号3实现的归约(即： $I \text{ if } S$)。

表5-9还为了移进而删除了移进-归约冲突。我们将表中的项目渐渐减少以显示出在何处发生了冲突。

表5-9 例5.12的SLR(1)分析表(删除了分析冲突)

状 态	输 入				Goto	
	if	Else	other	\$	S	I
0	s4		s3		1	2
1				接受		
2		r1		r1		
3		r2		r2		
4	s4		s3		5	2
5	s6		r3			
6	s4		s3		7	2
7		r4		r4		

5.3.3 SLR(1)分析能力的局限性

SLR(1)分析是LR(0)分析的一个简单但有效的扩展，而LR(0)分析的能力足以处理几乎所有实际的语言结构。不幸的是，在有些情况下，SLR(1)分析能力并不太强，而正由于这个原因，我们还需要学习更强大的一般的LR(1)和LALR(1)分析。下一个例子是SLR(1)分析失败的典型情况。

例5.13 考虑语句的以下文法，它是从Pascal中抽取和简化而得来的(在C中也有类似的情况

发生):

```

stmt    call-stmt | assign-stmt
call-stmt  identifier
assign-stmt  var := exp
var    var [ exp ] | identifier
exp    var | number

```

这个文法模块语句既可是调用无参数的过程也可以是对变量的表达式的赋值。请注意,赋值和过程调用都是以一个标识符开头。只有到看到语句的结尾或记号 `:=` 时,分析才会决定该语句是一个赋值还是一个调用。将这种情形简化成以下的文法,在其中删去了变量的替换选择,并将语句选项简化为无需改变基本的情况:

```

S    id | V := E
V    id
E    V | n

```

为了显示这个文法在 SLR(1) 分析中是如何引起一个分析冲突,请考虑项目集合的 DFA 的开始状态:

```

S    .S
S    .id
S    .V := E
V    .id

```

这个状态在 `id` 上有一个到状态

```

S    id.
V    id.

```

的转换。现在有 $\text{Follow}(S) = \{\$ \}$ 和 $\text{Follow}(V) = \{:=, \$ \}$ (由于有规则 $V \rightarrow V := E$ 所以有 `:=`, 又由于 E 可以是 V , 所以有 `$`)。因此, SLR(1) 分析算法要求在这个状态中有一个在输入符号 `$` 下的利用规则 $S \rightarrow id$ 和规则 $V \rightarrow id$ 实现的归约(这是一个归约-归约冲突)。这个分析冲突实际上是一个由 SLR(1) 方法的缺点所引起的“假冒”问题。实际上当输入为 `$` 时,用 $V \rightarrow id$ 实现的归约永远也不应该在这个状态中,这是由于只有到看到记号 `:=` 和被移进后,变量才会出现在语句的末端。

在下面的两节中,读者将会看到如何利用更强大的分析方法来解决这个分析问题。

5.3.4 SLR(k)文法

同其他分析算法一样, SLR(1) 分析算法可被扩展为 SLR(k) 分析,其中的分析动作是基于 $k-1$ 个先行的符号之上。利用上一章定义的集合 First_k 和 Follow_k , SLR(k) 分析程序使用以下两个规则:

1) 若状态 s 包含了格式 $A \rightarrow \cdot X$ (X 是一个记号), 且 $Xw \in \text{First}_k(X)$ 是输入串中之后的 k 个记号, 那么该动作就是将当前输入记号移进到栈中, 而且被压入到栈中的新状态是包含了项目 $A \rightarrow X \cdot$ 的状态。

2) 若状态 s 包含了完整项目 $A \rightarrow \cdot$, 且 $w \in \text{Follow}_k(A)$ 是输入串中之后的 k 个记号, 则动作利用规则 $A \rightarrow \cdot$ 归约。

当 $k > 1$ 时, SLR(k) 分析比 SLR(1) 分析更强大, 但由于分析表的大小将按 k 的指数倍增长,

所以它又要复杂许多。非SLR(1)的典型语言构造可利用LRLA(1)分析程序处理得更好一些,它可使用标准的消除二义性的规则,或将文法重写。虽然例5.13的简单非SLR(1)文法确实也可出现在SLR(2)中,但对于为任意值的 k 而言,它所来自的程序设计语言问题却不是SLR(k)。

5.4 一般的LR(1)和LALR(1)分析

本节将研究LR(1)分析的最一般格式,有时它称作LR(1)规范(canonical)分析。这种方法解决了上一节最后所提到的SLR(1)分析中出现的问题,但它却复杂得多。实际上在绝大多数情况下,通常地,一般的LR(1)分析太复杂以至于不能在大多数情况下的分析程序的构造中使用。幸运的是,一般的LR(1)分析的一个修正——称作LRLA(1)(即“先行”LR分析)在保留了LR(1)分析的大多数优点之外还保留了SLR(1)方法的有效性。LALR(1)方法已成为诸如用于诸如Yacc这样的分析程序生成器所选用的方法,本节稍后将会研究到它。但为了理解这个方法,首先应学习普通方法。

5.4.1 LR(1)项的有穷自动机

SLR(1)中的困难在于它在LR(0)项的DFA的构造之后提供先行,而构造却又忽略了先行。一般的LR(1)方法的功能在于它使用了一个从开始时就先将先行构建在它的构造中的新DFA。这个DFA使用了LR(0)项的扩展的项目。由于它们包括了每个项目中的一个先行记号,所以就称作**LR(1)项**(LR(1) item)。说得更准确一些就是:LR(1)项应是由LR(0)项和一个先行记号组成的对。利用中括号将LR(1)项写作

$$[A \quad . \quad a]$$

其中 $A \quad .$ 是一个LR(0)项,而 a 则是一个记号(先行)。

为了完成一般LR(1)分析所用的自动机的定义,我们需要首先定义LR(1)项之间的转换。它们与LR(0)转换相类似,但它们还知道先行。同LR(0)项一样,它们包括了 ϵ -转换,此外还需建立一个DFA,它的状态是项目为 ϵ -闭包的集合。LR(0)自动机和LR(1)自动机的主要差别在于 ϵ -转换的定义。我们首先给出较简单的情况下(非 ϵ -转换)的定义,它们与LR(0)的情况基本一致。

定义:LR(1)转换(第1部分)的定义(definition of LR(1) transitions (part 1))。假设有LR(1)项目 $[A \quad .X, a]$,其中 X 是任意符号(终结符或非终结符),那么 X 就有一个到项目 $[A \quad X, a]$ 的转换。

请注意在这种情形下,两个项目中都出现了相同的先行 a ,所以这些转换并不会引起新的先行的出现。只有 ϵ -转换才“创建”新的先行,如下所示。

定义:LR(1)转换(第2部分)的定义(definition of LR(1) transitions (part 2))。假设有LR(1)项目 $[A \quad .B, a]$,其中 B 是一个非终结符,那么对于每个产生式 $B \rightarrow$ 和在 $\text{First}(a)$ 中的每个记号 b 都有到项目 $[B \quad ., b]$ 的 ϵ -转换。

请读者留意,这些 ϵ -转换是如何跟踪在其中结构 B 需要被识别的上下文。实际上项目 $[A \quad .B, a]$ 说明了在分析的这一点上可能要识别 B ,但这只有是当这个 B 后跟有一个从串 a 衍生出的串时,且这样的串须以一个在 $\text{First}(a)$ 中的记号开始才可能。由于串 a 跟随在位于产生式 $A \rightarrow B$ 中的 B 之后,所以若 a 是被构造在 $\text{Follow}(A)$ 中,那么有 $\text{First}(a) \subseteq \text{Follow}(B)$,且在项目 $[B$

, b] 中的 b 将总是在 $\text{Follow}(B)$ 中。一般的 LR(1) 方法的功能在于集合 $\text{First}(a)$ 可能是 $\text{Follow}(B)$ 的一个恰当的子集 (SLR(1) 分析程序本质上从整个 Follow 集合中得到先行 b)。请读者再注意, 仅当可派生出空串时, 最初的先行 a 才可作为 b 中的一个元素。在大多数情况下 (尤其是在实际情况中), 只有当本身是 ϵ 时它才发生, 此时从格式 $[A \quad .B, a]$ 到 $[B \quad ., a]$ 可得到 ϵ - 转换的特殊情况。

对 LR(1) 项目集合的 DFA 的构造还包括指定开始状态。这是通过如同在 LR(0) 中一样用新的开始符号 S 和新的产生式 $S \rightarrow S$ (其中 S 是最初的开始符号) 来扩充文法。接着, LR(1) 项目的 NFA 的开始符号就成为了项目 $[S \quad .S, \$]$, 其中 $\$$ 代表结尾标记 (且是 $\text{Follow}(S)$ 中的唯一符号)。这就有效地说明了是由识别一个从 S 派生出的串开始, 其后则是 $\$$ 符号。

现在来看一下有关 LR(1) 项目的 DFA 的构造的一些示例。

例 5.14 考虑例 5.9 中的文法

$$A \rightarrow (A) \mid a$$

首先通过扩充文法以及构造初始的 LR(1) 项目 $[A \quad .A, \$]$ 来构建它的 LR(1) 项目集合的 DFA。这个项目的 ϵ - 闭包是 DFA 的开始状态。由于在这个项目中没有任何符号跟随在 A 之后 (按照前面所讨论的有关转换的术语, 串就是 ϵ), 就有到项目 $[A \quad .(A), \$]$ 和 $[A \quad .a, \$]$ 的 ϵ - 转换 (按照前面讨论过的有关术语, 即为 $\text{First}(\$) = \{\$ \}$)。接着, 开始状态 (状态 0) 就是这 3 个项目的集合:

$$\begin{aligned} \text{状态 0: } & [A \quad .A, \$] \\ & [A \quad .(A), \$] \\ & [A \quad .a, \$] \end{aligned}$$

在 A 上有从这个状态到包含了项目 $[A \quad .A, \$]$ 的集合的闭包的转换, 而且由于没有来自完整项目的转换, 所以这个状态仅包含了单个项目 $[A \quad .A, \$]$ 。除此之外再也没有来自这个状态的转换了, 所以将它编号为状态 1:

$$\text{状态 1: } [A \quad .A, \$]$$

(这是 LR(1) 分析算法将从中生成接受动作的状态)。

再回到状态 0 上, 在记号 (之上也有一个到由项目 $[A \quad .(A), \$]$ 组成的集合的闭包。由于有来自这个项目的 ϵ - 转换, 这个闭包也就并不是微不足道的了。实际上从这个项目上也有到项目 $[A \quad .(A),)]$ 和 $[A \quad .a,)]$ 的 ϵ - 转换。这是因为在项目 $[A \quad .(A), \$]$ 中, 在括号的上下文的右边识别 A 。也就是, 右边 A 的 Follow 是 $\text{First}(\$) = \{\$ \}$, 因此在这种情况下, 就得到了一个新的先行记号, 而且新的 DFA 状态是由以下的项目组成:

$$\begin{aligned} \text{状态 2: } & [A \quad .(A), \$] \\ & [A \quad .(A),)] \\ & [A \quad .a,)] \end{aligned}$$

再回到状态 0, 在其中找到由项目 $[A \quad .a, \$]$ 生成的状态的最后转换。由于这是一个完整的项目, 所以它是一个项目的状态:

$$\text{状态 3: } [A \quad .a, \$]$$

现在回到状态 2。在 A 上有一个从这个状态到 $[A \quad .(A), \$]$ 的 ϵ - 闭包的转换, 它是带有一个项目的状态:

$$\text{状态 4: } [A \quad .(A), \$]$$

在(上也有一个到 $[A \rightarrow (\cdot A),)]$ 的 ϵ -闭包的转换。在这里也可根据与在状态2的构造中相同的原理,生成闭包项目 $[A \rightarrow (\cdot A),)]$ 和 $[A \rightarrow \cdot a,)]$ 。所以就得到新的状态:

状态5: $[A \rightarrow (\cdot A),)]$
 $[A \rightarrow \cdot (A),)]$
 $[A \rightarrow \cdot a,)]$

请注意,这个状态除了第1个项目的先行之外,其他都与状态2相同。

最后,在记号a上有一个从状态2到状态6的转换:

状态6: $[A \rightarrow a\cdot,)]$

请读者再次注意,这与状态3几乎一样,只在先行上略有不同。

具有一个转换的下一个状态是状态4,它有一个在记号)上到状态

状态7: $[A \rightarrow (A)\cdot, \$]$

的转换。

再回到状态5,这里在(上有一个从这个状态到它本身的转换,还有一个在A上的到状态

状态8: $[A \rightarrow (A\cdot),)]$

的转换,以及一个在a上到早已构造好的状态6的转换。

最后,在)上有一个从状态8到

状态9: $[A \rightarrow (A)\cdot,)]$

的转换。

因此,这个文法的LR(1)项目的DFA具有10个状态。图5-7是完整的DFA。将它与相同文法的LR(0)项目集合的DFA相比较(参见图5-5),我们发现LR(1)项目的DFA几乎是它的两倍大小。这也是正常的。实际上通过在带有多个先行记号的复杂情形中的10因子,LR(1)状态的数量可超过LR(0)状态的数量。

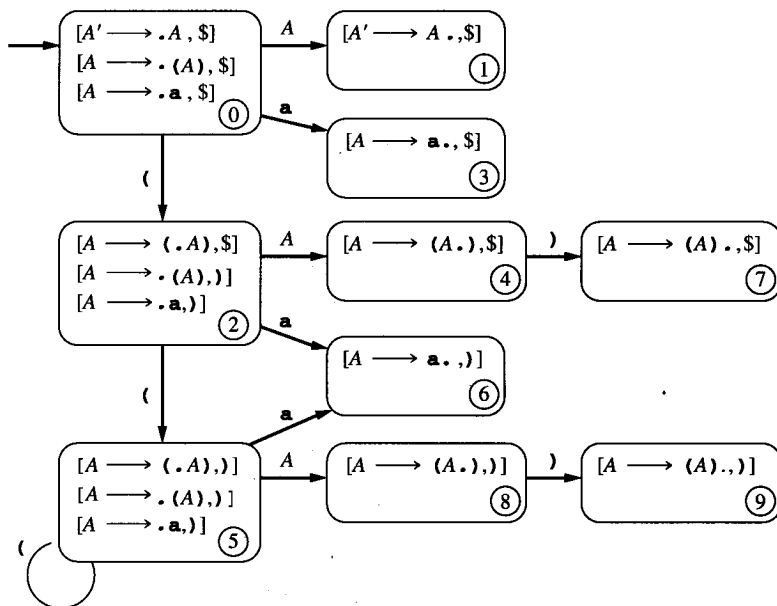


图5-7 例5.14的LR(1)项目集合的DFA

5.4.2 LR(1)分析算法

在考虑其他示例之前，我们需要先根据新的 DFA 构造通过重新叙述分析算法而将一般的 LR(1) 分析进行完善。由于仅需重述 SLR(1) 分析算法而无需使用 LR(1) 项目中的先行记号代替 Follow 集合，所以这是容易办到的。

一般的 LR(1) 分析算法 令 s 为当前状态(位于分析栈的顶部)，则动作定义如下：

若状态 s 包含了格式 $[A \quad .X, a]$ 的任意 LR(1) 项目，其中 X 是一个终结符且是输入串中的下一个记号，则动作就是将输入记号移进到栈中，且被压入到栈中的新状态是包含了 LR(1) 项目 $[A \quad X, a]$ 的状态。

若状态 s 包含了完整的 LR(1) 项目 $[A \quad \cdot, a]$ ，且输入串中的下一个记号是 a ，则动作就是用规则 $A \rightarrow$ 归约。用规则 $S \rightarrow S$ (其中 S 是开始状态) 实现的归约等价于接受(只有当下一个输入记号是 $\$$ 时才发生)。在其他情况下，新状态的计算如下：将串 X 以及与其对应的所有状态从分析栈中删去。相应地 DFA 返回到开始构造的状态。通过构造，这个状态必须包括格式 $[B \quad \cdot A, b]$ 的 LR(1) 项目。将 A 压入到栈中，并压入包含了项目 $[B \quad \cdot A, b]$ 的状态。

若下一个输入记号不是上面所述的任何一种情况，则声明一个错误。

同使用前面的方法一样，若前面的一般 LR(1) 分析规则的应用程序不引起二义性，则该文法就是 LR(1) 文法(LR(1) grammar)。特别地，当且仅当对于任何状态 s ，能够满足以下两个条件，该文法才是 LR(1) 文法：

对于在 s 中的任何项目 $[A \quad .X, a]$ ，且 X 是一个终结符，则在 s 中没有格式 $[B \quad \cdot, X]$ 的项目(否则就有一个移进-归约冲突)。

在 s 中没有格式 $[A \quad \cdot, a]$ 和 $[B \quad \cdot, a]$ 的两个项目(否则就有一个归约-归约冲突)。

我们还注意到可从表达一般 LR(1) 分析算法的 LR(1) 项目集合的 DFA 中构造出一个分析表。该表具有与 SLR(1) 分析程序的表格完全相同的格式，如下例所示。

例 5.15 为表 5-10 中的例 5.14 的文法给出一般的 LR(1) 分析表。它可从图 5-7 的 DFA 中很方便地构造出来。在该表中，我们为在归约动作中的文法规则选择使用了以下的编号：

(1) $A \rightarrow (A)$

(2) $A \rightarrow a$

因此在状态 3 中带有先行 $\$$ 的项 r_2 指出了规则 $A \rightarrow a$ 实现的归约。

表 5-10 例 5.14 的一般 LR(1) 分析表

状 态	输 入					Goto
	(a)	\$	A	
0	s2	s3			1	
1					接受	
2	s5	s6				4
3					r2	
4			s7			
5	s5	s6				8
6			r2			
7				r1		
8			s9			
9			r1			

因为在一般的LR(1)分析中的特定串的分析步骤与它们在SLR(1)分析或LR(0)分析中的步骤相同,所以我们省略了一个这样的分析示例。由于也可从DFA中很方便地得到,所以本节随后的例子也将省掉分析表的构造。

在实际中除非是有二义性的,几乎所有合理的程序设计语言的文法都是LR(1)文法。当然也有可能构造出不能成为LR(1)文法的非二义性文法的示例来,但在这里就不这样做了(参见练习)。在实践中人们总是试图避免它。实际上,程序设计语言甚至很少需要一般的LR(1)分析所提供的能力。我们前面用来介绍LR(1)分析的示例(例5.14)其实早已是一个LR(0)文法了(而且因此也就是SLR(1)文法了)。

下面的示例表明一般的LR(1)分析解决了例5.13中不能成为SLR(1)的文法的先行问题。

例5.16 例5.13的文法在简化了的格式中是:

$$\begin{aligned} S & \quad id \mid V := E \\ V & \quad id \\ E & \quad V \mid n \end{aligned}$$

为这个文法构造LR(1)项目集合的DFA。其开始状态是项目 $[S \quad .S, \$]$ 的闭包,它包括了项目 $[S \quad .id, \$]$ 和 $[S \quad .V := E, \$]$ 。因为 $S \quad .V := E$ 指出可能识别了一个 V ,但是这只有是当它后面是一个赋值记号时,所以这个最后的项目还引起了闭包项目 $[V \quad .id, :=]$ 。因此,记号 $:=$ 作为初始项目 $V \quad .id$ 的先行出现。开始状态可总结为由以下的LR(1)项目组成:

$$\begin{aligned} \text{状态0:} \quad & [S \quad .S, \$] \\ & [S \quad .id, \$] \\ & [S \quad .V := E, \$] \\ & [V \quad .id, :=] \end{aligned}$$

在 S 上有一个从这个状态到一个项目的状态

$$\text{状态1:} \quad [S \quad .S, \$]$$

的转换,且在 id 上有一个到两个项目的状态

$$\begin{aligned} \text{状态2:} \quad & [S \quad .id, \$] \\ & [V \quad .id, :=] \end{aligned}$$

的转换。在 V 上还有一个从状态0到一个项目的状态

$$\text{状态3:} \quad [S \quad .V := E, \$]$$

的转换。从状态1和状态2没有转换,但在 $:=$ 上从状态3到项目 $[S \quad V := .E, \$]$ 的闭包有一个转换。由于在这个项目中 E 之后没有符号,所以这个闭包包括了项目 $[E \quad .V, \$]$ 和 $[E \quad .n, \$]$ 。最后,由于此时 V 之后也没有符号,项目 $[E \quad .V, \$]$ 可引起闭包项目 $[V \quad .id, \$]$ (这与在状态0中的情形不同,在状态0中 V 后有一个赋值。而由于此时已经看到了一个赋值记号,所以这里的 V 后不能有一个赋值)。那么最后的状态就是

$$\begin{aligned} \text{状态4:} \quad & [S \quad V := .E, \$] \\ & [E \quad .V, \$] \\ & [E \quad .n, \$] \\ & [V \quad .id, \$] \end{aligned}$$

后面的状态和转换能够很容易地构造出来,我们将它留给读者。图5-8中是LR(1)项目集合的完

整的DFA。

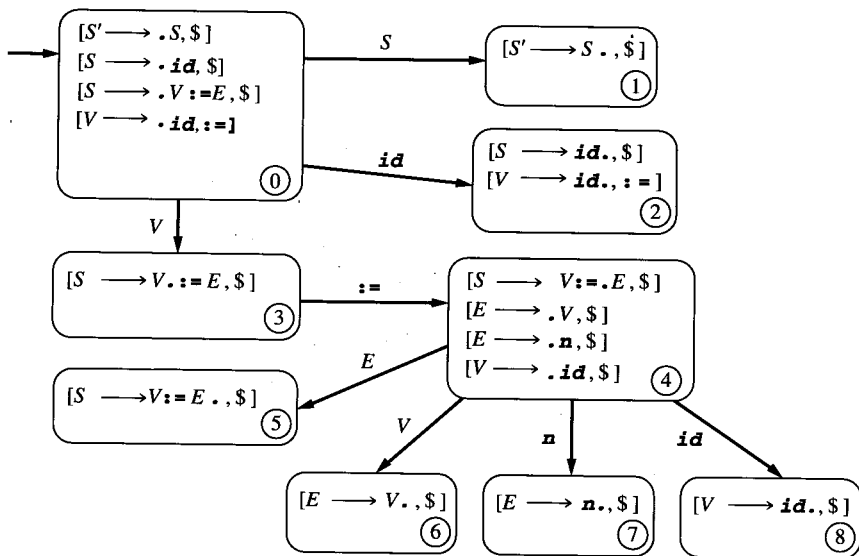


图5-8 例5.16中LR(1)项目集的DFA

现在来考虑状态2。这是引起SLR(1)分析冲突的状态。LR(1)项目可由它的先行清晰地区分出两个归约：在 $:=$ 上用规则 $S \rightarrow id$ 实现的归约和用 $V \rightarrow id$ 实现的归约。因此这个文法就是LR(1)文法。

5.4.3 LALR(1)分析

LALR(1)分析是基于以下的观察：在许多情况下，LR(1)项目集的DFA的大小应部分地归于许多不同状态的存在，在这些状态的项目(LR(0)项目)中，第1个成分的集合相同，只有第2个成分(先行符号)不同。例如，图5-7的LR(1)项目的DFA有10个状态，而相应的LR(0)项目的DFA(图5-5)只有6个。实际上在图5-7的状态2和5、状态4和8、状态7和9、状态3和6中，每对中的状态与其他的区别只在于它的项目的先行成分。例如状态2和5：这两个状态只在其第1个项目上不同，且仅在那个项目的先行上：状态2有第1个项目 $[A \rightarrow (.A), \$]$ ，且 S 作为它的先行；而状态5有第1个项目 $[A \rightarrow (.A),)]$ ，且 $)$ 作为它的先行。

LALR(1)分析算法表明了它使得标识所有这样的状态和组合它们的先行有意义。在这样做时，我们总是必须以一个与LR(0)项目中的DFA相同的DFA作为结尾，但是每个状态都是以带有先行集合的项目组成。在完整项目的情况下，这些先行集合通常比相应的Follow集合小；因此，LRLA(1)分析保留了LR(1)分析优于SLR(1)分析的一些特征，但是仍具有在LR(0)项目中的DFA尺寸较小的特点。

更正式地，在LR(1)项目的DFA中，状态的核心(core)是由状态中的所有LR(1)项目的第1个成分组成的LR(0)项目的集合。由于LR(1)项目的DFA的构造使用与在LR(0)项目的DFA的构造中相同的转换，但是它们在项目的先行部分上的作用不同，我们得到以下两个事实，它们构成了LALR(1)分析构造的基础。

(1) LALR(1)分析的第1个原则

LR(1)项目的DFA的状态核心是LR(0)项目的DFA的一个状态。

(2) LALR(1)分析的第2个原则

若有具有相同核心的LR(1)项目的DFA的两个状态 s_1 和 s_2 ,假设在符号 X 上有一个从 s_1 到状态 t_1 的转换,那么在 X 上就还有一个从状态 s_2 到一个状态 t_2 的转换,且状态 t_1 和 t_2 具有相同的核。

总之,这两个原则允许我们构造LALR(1)项目的DFA(DFA of LALR(1) items),它是通过识别具有相同核心的所有状态以及为每个LR(0)项目构造出先行符号的并,而从LR(1)项目的DFA构造出来。因此,这个DFA的每个LALR(1)项目都将一个LR(0)项目作为它的第1个成分,并将一个先行记号的集合作为它的第2个成分 \ominus 。在后面的示例中将在先行之间写一个/来表示多重先行。因此,LALR(1)项目 $[A \rightarrow \cdot, a/b/c]$ 具有一个由符号 a 、 b 和 c 组成的先行集合。

我们给出一个展示这个构造的示例。

例5.17 考虑例5.14的文法,它的LR(1)项目的DFA在图5-7中。通过识别状态2和5、状态4和8、状态7和9、状态3和6,就可得出图5-9中的LALR(1)项目的DFA。在那个图中,我们保留了状态2、3、4和7的编号,并从状态5、6、8和9添加了先行。正如所期望的,除了先行之外,这个DFA与LR(0)项目的DFA相同。

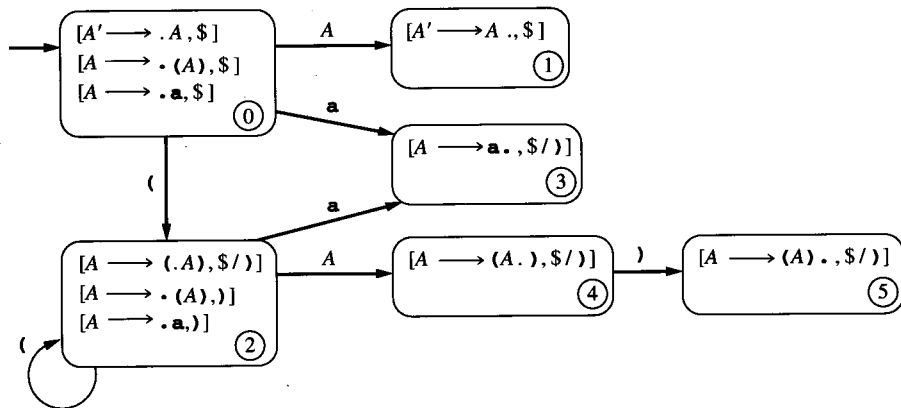


图5-9 例5.17的LALR(1)项目集合的DFA

使用了LALR(1)项目的压缩了的DFA的LALR(1)分析算法与上一节所描述的一般的LR(1)分析算法相同。同前所述,若在LALR(1)分析算法中没有出现分析冲突则称这个文法为LALR(1)文法(LALR(1) grammar)。在LALR(1)构造中,有可能制造出在一般的LR(1)分析中不存在的分析冲突来,但这在实际中却很少发生。实际上,若一个文法是LR(1)文法,则LALR(1)分析表就不能有任何的移进-归约冲突了;但是却有可能有归约-归约冲突(参见练习)。然而,若一个文法是SLR(1),那么它肯定就是LALR(1)文法,而且LALR(1)分析程序在消除发生在SLR(1)分析中的典型冲突时通常与一般的LR(1)分析程序所做的相同。例如,例5.16的非SLR(1)文法是LALR(1):图5-8的LR(1)项目的DFA也是LALR(1)项目的DFA。如在这个示例中一样,如果文法已经是LALR(1)了,则使用LALR(1)分析取代一般的LR分析的唯一结果是:在声明错误之前会作出一些虚假的归约。例如,从图5-9中可看出,若有错误的输入串 a ,则LALR(1)分析程序将在声明错误之前执行归约 $A \rightarrow a$,但是一般的LR(1)分析程序将在

\ominus LR(1)项目的DFA实际上还可以利用先行符号的集合代表共享它们的第1个成分的相同状态中的多重项目,但是我们发现为LALR(1)构造使用这种表示法也很方便,在这里它最适合。

移进记号a之后立即声明错误。

将LR(1)状态组合到LALR(1)项目的DFA解决了分析表尺寸较大的问题,但它仍要求计算LR(1)项目的整个DFA。实际上,通过传播先行(propagating lookahead)的处理从LR(0)项目的DFA直接计算出LALR(1)项目的DFA是有可能的。尽管我们对此并不着意进行描述,但看看如何相对简单地做到它仍是不无好处的。

观察图5-9中的LALR(1)DFA。首先通过将结尾标记\$添加到状态0中的扩充项目 $A \rightarrow A$ 的先行中(\$先行被称作被自发生成的(spontaneously generated))。接着再通过 ϵ -闭包的规则,将\$传播到两个闭包项目(核心项目 $A \rightarrow A$ 的右边的A后是空串)。通过跟随状态0的3个转换,将\$传播到状态1、3和2的核心项目。继续状态2,闭包项目得到先行),再一次通过自发生成(因为位于核心项目 $A \rightarrow (A)$ 上的A在右括号之前)。现在在a上的到状态3的转换使得将)传送到那个状态中项目的先行(上的从状态2到它本身的转换也使得)。传送到核心项目的先行(这就是为什么核心项目在它的先行集合中有\$和的原因)。现在先行集合\$/传送到状态4,之后再到了状态7。因此通过这个处理就可直接从LR(0)项目的DFA得到图5-9中的LALR(1)的DFA了。

5.5 Yacc : 一个LALR(1)分析程序的生成器

分析程序生成器(parser generator)是一个指定某个格式中的一种语言的语法作为它的输入,并为该种语言产生分析过程以作为它的输出的程序。在历史上,分析程序生成器被称作编译-编译程序(compiler-compiler),这是由于按照规律可将所有的编译步骤作为包含在分析程序中的动作来执行。现在的观点是将分析程序仅考虑为编译处理的一个部分,所以这个术语也就有些过时了。合并LALR(1)分析算法是一种常用的分析生成器,它被称作Yacc(yet another compiler-compiler)。本节将给出Yacc的概貌来,下一节将使用Yacc为TINY语言开发一个分析程序。由于Yacc有许多不同的实现以及存在着通常称作Bison的若干个公共领域版本^①,所以在它的运算细节中有许多变化,而这又可能与这里所用的版本有些不同^②。

5.5.1 Yacc基础

Yacc取到一个说明文件(通常带有一个.y后缀)并产生一个由分析程序的C源代码组成的输出文件(通常是在一个称作y.tab.c或ytab.c或更新一些的<文件名>.tab.c的文件中,而<文件名>.y则是输入文件)。Yacc说明文件具有基本格式

```
{definitions}
%%
{rules}
%%
{auxiliary routines}
```

因此就有3个被包含了双百分号的行分隔开来的部分——定义部分、规则部分和辅助程序部分。

定义部分包括了Yacc需要用来建立分析程序的有关记号、数据类型以及文法规则的信息。它还包括了必须在它的开始时直接进入输出文件的任何C代码(主要是其他源代码文件的#include指示)。说明文件的这个部分可以是空的。

① 一个流行版本——Gnu Bison——是由Free Software Foundation发布的Gnu软件的一个部分,请参见“注意与参考”部分。

② 实际上,我们已使用了若干个不同的版本来生成后面的示例。

规则部分包括修改的BNF格式中的文法规则以及将在识别出相关的文法规则时被执行的C代码中的动作(即:根据LALR(1)分析算法,在归约中使用)。文法规则中使用的元符号惯例如下:通常,竖线被用作替换(也可分别写出替换项)。用来分隔文法规则的左右两边的箭头符号在Yacc中被一个冒号取代了,而且必须用分号来结束每个文法规则。

第3点:辅助程序部分包括了过程和函数声明,除非通过#include文件,否则它们会不适用,此外还需要被用来完成分析程序和/或编译程序。这个部分也可为空,此时第2个双百分号元符号可从说明文件中省略掉。因此,最小的Yacc说明文件可仅由后面带有文法规则和动作(若仅是要分析文法也可省掉动作,本节稍后再讲到它)的%%组成。

Yacc还允许将C风格的注解插入到说明文件的任何不妨碍基本格式的地方。

利用一个简单的示例,我们可将Yacc说明文件的内容解释得更详细一些。这个示例是带有文法

```
exp    exp addop term | term
addop  + | -
term    term mulop factor | factor
mulop   *
factor  ( exp ) | number
```

的简单整型算术表达式的计算器。这个文法在前面的章节中已用得很多了。在4.1.2节中,我们为这个文法开发了一个递归下降计算器程序。程序清单5-1给出了完全等价的Yacc说明。下面按顺序讨论这个说明中3个部分的内容。

程序清单5-1的定义部分有两个项目。第1个项目由要在Yacc输出的开始处插入的代码组成。这个代码是由两个典型的#include指示组成,且由在其前后的分隔符%{和%}将它与这个部分中的其他Yacc说明分开(请注意百分号在括号之前)。定义部分的第2个项目是记号NUMBER的声明,它代表数字的一个序列。

程序清单5-1 一个简单计算器程序的Yacc定义

```
%{
#include <stdio.h>
#include <ctype.h>
%}

%token NUMBER

%%

command : exp      { printf("%d\n",$1);}
        ; /* allows printing of the result */

exp      : exp '+' term  {$$ = $1 + $3;}
        | exp '-' term  {$$ = $1 - $3;}
        | term           {$$ = $1;}
        ;

term      : term '*' factor {$$ = $1 * $3;}
        | factor           {$$ = $1;}
        ;
```

```

factor      : NUMBER      {$$ = $1;}
            | '(' exp ')' {$$ = $2;}
            ;

%%

main()
{ return yyparse();
}

int yylex(void)
{ int c;
  while((c = getchar()) == ' ');
  /* eliminates blanks */
  if ( isdigit(c) ) {
    ungetc(c,stdin);
    scanf("%d",&yylval);
    return(NUMBER);
  }
  if (c == '\n') return 0;
  /* makes the parse stop */
  return(c);
}

void yyerror(char * s)
{ fprintf(stderr,"%s\n",s);
} /* allows for printing of an error message */

```

Yacc用两种方法来识别记号。首先，文法规则的单引号中的任何字符都可被识别为它本身。因此，单字符记号就可直接被包含在这个风格的文法规则中，正如程序清单 5-1中的运算符记号+、-和*（以及括号记号）。其次，可在Yacc的%记号(%token)中声明符号记号，如程序清单 5-1中的记号NUMBER。这样的记号被Yacc赋予了不会与任何字符值相冲突的数字值。典型地，Yacc开始用数字258给记号赋值。Yacc将这些记号定义作为#define语句插入到输入代码中。因此，在输出文件中就可能会找到行

```
#define NUMBER 258
```

作为Yacc对说明文件中的%token NUMBER声明的对应。Yacc坚持定义所有的符号记号本身，而不是从别的地方引入一个定义。但是却有可能通过在记号声明中的记号名之后书写一个值来指定将赋给记号的数字值。例如，写出

```
%token NUMBER 18
```

就将给NUMBER赋值18（不是258）。

在程序清单5-1的规则部分中，我们看到非终结符exp、term和factor的规则。由于还需要打印出一个表达式的值，所以还有另外一个称为command的规则，而且将其与打印动作相结合。因为首先列出了command的规则，所以command则被作为文法的开始符号。若不这样，我们还可再在定义部分中包括行

```
%start command
```

此时就不必将command的规则放在开头了。

Yacc中的动作是由在每个文法规则中将其写作真正的C代码（在花括号中）来实现的。通常，尽管也有可能在一个选择中写出嵌入动作（embedded action）（稍后将讨论它），但动作代码仍是放在每个文法规则选择的末尾（但在竖线或分号之前）。在书写动作时，可以享受到Yacc伪变量

(pseudovariable)的好处。当识别一个文法规则时,规则中的每个符号都拥有一个值,除非它被参数改变了,该值将被认为是一个整型(稍后将会看到这种情况)。这些值由Yacc保存在一个与分析栈保持平行的值栈(value stack)中。每个在栈中的符号值都可通过使用以\$开始的伪变量来引用。\$\$代表刚才被识别出来的非终结符的值,也就是在文法规则左边的符号。伪变量\$1、\$2、\$3等等都代表了文法规则右边的每个连续的符号。因此在程序清单 5-1中,文法规则和动作

```
exp      : exp '+' term { $$ = $1 + $3; }
```

就意味着当识别规则 *exp* *exp* + *term*时,就将左边的*exp* 的值作为*exp* 的值与右边的*term* 的值之和。

所有的非终结符都是通过这样的用户提供的动作来得到它们的值。记号也可被赋值,但这是在扫描过程中实现的。Yacc假设记号的值被赋给了由Yacc内部定义的变量yylval,且在识别记号时必须给yylval赋值。因此,在文法和动作

```
factor   : NUMBER      { $$ = $1; }
```

中,值\$1指的是当识别记号时已在前面被赋值为yylval的NUMBER记号的值。

程序清单5-1的第3个部分(辅助程序部分)包括了3个过程的定义。第1个是main的定义,之所以包含它是因为Yacc输出的结果可被直接编译为可执行的程序。过程main调用yyparse,yyparse是Yacc给它所产生的分析过程起的名称。这个过程被声明是返回一个整型值。当分析成功时,该值总为0;当分析失败时,该值为1(即发生一个错误,且还没有执行错误恢复)。Yacc生成的yyparse过程接着又调用一个扫描程序过程,该过程为了与Lex扫描程序生成器相兼容,所以就假设叫作yylex(参见第2章)。因此,程序清单5-1中的Yacc说明还包括了yylex的定义。在这个特定的情况下,yylex过程非常简单。它所需要做的只有返回下一个非空字符;但若这个字符是一个数字,此时就必须识别单个元字符记号NUMBER并返回它在变量yylval中的值。这里有一个例外:由于假设一行中输入了一个表达式,所以当扫描程序已到达了输入的末尾时,输入的末尾将由一个新行字符(在C中的'\n')指出。Yacc希望输入的末尾通过yylex由空值0标出(这也是Lex所共有的一个惯例)。最后就定义了一个yyerror过程。当在分析时遇到错误时,Yacc就使用这个过程打印出一个错误信息(典型地,Yacc打印串“语法错误”,但这个行为可由用户改变)。

5.5.2 Yacc选项

除了yylex和yyerror之外,Yacc通常需要访问许多辅助过程,而且它们经常是被放在外置的文件之中而非直接放在Yacc说明文件中。通过写出恰当的头文件以及将#include指示放在Yacc说明的定义部分中,就可以很容易地使Yacc访问到这些过程。将Yacc特定的定义应用到其他文件上就要复杂一些了,在记号定义时尤为如此。此时正如前面所讲的,Yacc坚持自己生成(而不是引入),但是它又必须适用于编译程序的许多其他部分(尤其是扫描程序)。正是由于这个原因,Yacc就有一个可用的选项,自动产生包含了该信息的头文件,而这个头文件将被包括在需要定义的任何其他文件中。这个头文件通常叫作y.tab.h或ytab.h,并且它与-d选项(用于header文件)一起生成。

例如,若文件calc.y包括了程序清单5-1中的Yacc说明,则命令

```
yacc -d calc.y
```

将产生(除了文件y.tab.c文件之外)内容各异的文件y.tab.h(或相似的名称),但却总是包括

下示的内容：

```
#ifndef YYSTYPE
#define YYSTYPE int
#endif
#define      NUMBER      258
extern YYSTYPE yylval;
```

(稍后将详细地描述YYSTYPE的含义)。这个文件可被用来通过插入行

```
#include y.tab.h
```

到文件中而将yylex的代码放在另一个文件中^①。

Yacc的另一个且极为有用的选项是详细选项(verbose option)，它是由命令行中的-v标志激活。这个选项也产生另一个名字为y.output(或类似名称的)的文件。这个文件包括了被分析程序使用的LALR(1)分析表的文本描述。阅读这个文件将使得用户可以准确地判断出Yacc生成的分析程序在任何情况下将会采取的动作，而且这是处理文法中的二义性和不准确性的极为有效的方法。在向说明添加动作或辅助过程之前，Yacc与这个选项一起在文法上运行以确保Yacc生成的分析程序将确实如希望的那样执行的确是一个好办法。

例如，程序清单5-2中的Yacc说明。这是程序清单5-1中Yacc说明的基本版本。当同时使用Yacc和详细选项：

```
yacc -v calc.y
```

程序清单5-2 使用-V选项的Yacc说明提纲

```
%token NUMBER
%%
command      : exp
              ;
exp           : exp '+' term
              | exp '-' term
              | term
              ;
term          : term '*' factor
              | factor
              ;
factor        : NUMBER
              | '(' exp ')'
```

时，这两个说明生成相同的输出文件。程序清单5-3是该文法完整的典型y.output文件^②。下面的段落将讨论如何解释这个文件。

Yacc输出文件列出了DFA中的所有状态，此外还有内部统计的小结。状态由0开始编号。输出文件在每个状态的下面列出了核心项目(并未列出闭包项目)，其次是与各个先行对应的动作，最后则是各个非终结符的goto动作。Yacc尤其使用一个下划线字符_来标出项目中的显著

① Yacc的早期版本可能只将记号的定义(但没有yylval的定义)放置在y.tab.h中。这可能将要求一个共同的工作区或重新安排代码。

② Bison的较新版本在输出文件中产生了一个根本不同的格式，但是内容却基本相同。

的位置，用它来代替本章所用到的句点，却用句点来指明缺省，或“不在意”每个状态列表的动作部分中的先行记号。

程序清单5-3 为程序清单5-1中的Yacc说明使用详细选项生成的典型y.output 文件

```

state 0
    $accept : _command $end

    NUMBER shift 5
    ( shift 6
    . error

    command goto 1
    exp goto 2
    term goto 3
    factor goto 4

state 1
    $accept : command_$end

    $end accept
    . error

state 2
    command : exp_ (1)
    exp : exp_+ term
    exp : exp_- term

    + shift 7
    - shift 8
    . reduce 1

state 3
    exp : term_ (4)
    term : term_* factor

    * shift 9
    . reduce 4

state 4
    term : factor_ (6)

    . reduce 6

state 5
    factor : NUMBER_ (7)

    . reduce 7

state 6
    factor : (_exp )

    NUMBER shift 5
    ( shift 6
    . error

    exp goto 10
    term goto 3
    factor goto 4

state 7
    exp : exp+_term

    NUMBER shift 5
    ( shift 6
    . error

    term goto 11
    factor goto 4

state 8
    exp : exp-_term

    NUMBER shift 5
    ( shift 6
    . error

    term goto 12
    factor goto 4

state 9
    term : term*_factor

    NUMBER shift 5
    ( shift 6
    . error

    factor goto 13

state 10
    exp : exp_+ term
    exp : exp_- term
    factor : ( exp_)

    + shift 7
    - shift 8
    ) shift 14
    . error

state 11
    exp : exp + term_ (2)
    term : term_* factor

    * shift 9
    . reduce 2

```

```
state 12
  exp : exp - term_ (3)
  term : term_ * factor
```

```
* shift 9
. reduce 3
```

```
state 13
```

```
term : term * factor_ (5)
```

```
. reduce 5
```

```
state 14
```

```
factor : ( exp )_ (8)
```

```
. reduce 8
```

```
8/127 terminals, 4/600 nonterminals
9/300 grammar rules, 15/1000 states
0 shift/reduce, 0 reduce/reduce conflicts reported
9/601 working sets used
memory: states, etc. 36/2000, parser 11/4000
9/601 distinct lookahead sets
6 extra closures
18 shift entries, 1 exceptions
8 goto entries
4 entries saved by goto default
Optimizer space used: input 50/2000, output 218/4000
218 table entries, 202 zero
maximum spread: 257, maximum offset: 43
```

Yacc通过列出扩充产生式的初始项目而从状态 0 开始，而这通常在 DFA 的开始状态中只有核心项目。在上面示例的输出文件中，这个项目写作

```
$accept : _command $end
```

它与我们自己的术语中的项目 *command* 对应。Yacc 为扩充的非终结符提供的名字为 *\$accept*。它还将输入结尾的伪记号显式地列为 *\$end*。

首先大致地看一下状态 0 的动作部分，它后面是核心项目的列表：

```
NUMBER shift 5
( shift 6
. error
command goto 1
exp goto 2
term goto 3
factor goto 4
```

上面的列表指出了 DFA 移进到先行记号 *NUMBER* 的状态 5 中、移进到先行记号 (的状态 6 中，并且说明其他所有先行记号中的错误。此外为了在归约到所给出的非终结符中使用还列出了 4 个 *goto* 转换。这些动作与用这章中的方法手工构造分析表中的内容完全一样。

再看看状态 2，它有输出列表

```
state 2
command : exp_ (1)
exp : exp_ + term
exp : exp_ - term
+ shift 7
- shift 8
. reduce 1
```

这里的核心项目是一个完整的项目，所以在动作部分中有一个用相关的产生式选择实现的归约。为了提醒我们在归约中所使用的产生式的编号，Yacc 在完整的项目之后列出了编号。在这种情

况下，产生式编号就是 1，而且有一个表示用产生式 *command* *exp* 实现的 **reduce 1** 动作。Yacc 总是按照它们在说明文件中所列的顺序为产生式编号。在我们的示例中，有 8 个产生式 (*command* 的一个，*exp* 的 3 个，*term* 的两个以及 *factor* 的两个)。

请注意，在这个状态中的归约动作是一个缺省动作：一个将在任何不是 + 或 - 的先行之上的归约。这里的 Yacc 与一个单纯的 LALR(1) 分析程序 (而且甚至是 SLR(1) 分析程序) 的不同在于它并不试着去检查归约上的合法先行 (而是在若干个归约中决定)。Yacc 分析程序将在最终声明错误之前在错误之上作出多个归约 (这将在任何的更多的移进发生之前的最后行为)。这就意味着错误信息可能不是像它们应该的那样有信息价值，但是分析表却会变得十分简单，这是因为情况发生得更少了 (这点将在 5.7 节中再次讨论到)。

在这个示例最后，我们从 Yacc 输出文件中构造出一个分析表，该表与本章早些时候手工写出的完全一样。表 5-11 就是这个分析表。

表 5-11 与程序清单 5-3 的 Yacc 输出对应的分析表

状 态	输 入							Goto			
	NUMBER	(+	-	*)	\$	<i>command</i>	<i>exp</i>	<i>term</i>	<i>factor</i>
0	s5	s6						1	2	3	4
1							accept				
2	r1	r1	s7	s8	r1	r1	r1				
3	r4	r4	r4	r4	s9	r4	r4				
4	r6	r6	r6	r6	r6	r6	r6				
5	r7	r7	r7	r7	r7	r7	r7				
6	s5	s6							10	3	4
7	s5	s6								11	4
8	s5	s6								12	4
9	s5	s6									13
10			s7	s8		S14					
11	r2	r2	r2	r2	s9	r2	r2				
12	r3	r3	r3	r3	s9	r3	r3				
13	r5	r5	r5	r5	r5	r5	r5				
14	r8	r8	r8	r8	r8	r8	r8				

5.5.3 分析冲突与消除二义性的规则

详细选项的一个重要用处是 Yacc 将在 **y.output** 文件中报告调查的分析冲突。Yacc 在其中建立了消除二义性的规则，该规则将允许它甚至在分析冲突发生时产生一个分析程序 (因此，甚至这也是用于二义性文法的)。这些消除二义性的规则通常总能做对，但有时也会出错。对 **y.output** 文件的检查使用户判断出分析冲突是什么，以及由 Yacc 产生的分析程序是否可正确地解决问题。

程序清单5-1的示例中并没有分析冲突,在输出文件结尾的小结信息中,Yacc将它报告为

```
0 shift / reduce, 0 reduce / reduce conflicts reported
```

例5.12中的二义性悬挂else文法是一个更为有趣的示例。在表5-9中,我们为这个文法给出了SLR(1)分析表,在其中通过选取移进而不是归约将状态5中的移进-归约冲突消除了(它与最近嵌套消除二义性的规则对应)。Yacc以完全相同的术语报告了二义性,并且通过相同的消除二义性的规则解决了二义性。除了Yacc插入到错误项中的缺省归约之外,由Yacc报告的分析表确实与表5-9相同。例如,Yacc将状态5的动作在y.output文件中报告如下(记号在说明文件中的定义用小写字母写出,这样就避免了与C的保留字相冲突):

```
5: shift / reduce conflict ( shift 6'n red) on ELSE
state 5
    I : IF S_(3)
    I : IF S_ELSE S
    ELSE shift 6
    . reduce 3
```

在小结信息中,Yacc还报告了一个移进-归约冲突:

```
1 shift / reduce, 0 reduce / reduce conflicts reported
```

在归约-归约冲突的情况下,Yacc通过执行由文法规则在说明文件中首先列出的归约来消除二义性。尽管它也会带来正确的分析程序,这仍与在文法中的错误很相似。下面是一个简单的示例。

例5.18 考虑以下的文法:

$$\begin{aligned} S & A \mid B \\ A & a \\ B & a \end{aligned}$$

由于单个合法串 a 有两个派生: $S \Rightarrow A \Rightarrow a$ 和 $S \Rightarrow B \Rightarrow a$,所以这是一个有二义性的文法。程序清单5-4是这个文法完整的y.output文件。请注意,在状态4中的归约-归约冲突,它由在规则 $B \Rightarrow a$ 之前执行规则 $A \Rightarrow a$ 来解决。这样就导致了后面的这个规则永远不会用在归约之中(它很明确地指出文法的一个问题)。Yacc在最后报告了这项情况以及行

```
Rule not reduced : B : a
```

程序清单5-4 例5.18中文法的Yacc输出文件

```
Rule not reduced: B : a

state 0
    $accept : _S $end

    a shift 4
    . error

    S goto 1
    A goto 2
    B goto 3

state 1
    $accept : S_$end
```

```

$end accept
. error

state 2
S : A_ (1)

. reduce 1

state 3
S : B_ (2)

. reduce 2

4: reduce/reduce conflict (red'ns 3 and 4 ) on $end
state 4
A : a_ (3)
B : a_ (4)

. reduce 3

Rule not reduced: B : a

3/127 terminals, 3/600 nonterminals
5/300 grammar rules, 5/1000 states
0 shift/reduce, 1 reduce/reduce conflicts reported
...

```

除了前面已提到过的消除二义性的规则之外，Yacc为指定与一个有二义性的文法相分隔开的算符优先及结合性还具有特别的机制。它具有一些优点。首先，文法无需包括指定了结合性和优先权的显式构造，而这就意味着文法可以短一些和简单一些了。其次，相结合的分析表也可小一点且得出的分析程序更有效。

例如程序清单 5-5 中的 Yacc 说明。在那个图中，文法是用没有算符的优先权和结合性的具有二义性的格式书写。相反地，算符的优先权和结合性通过写出行

```

%left '+' '-'
%left '*'

```

在定义部分中给出来。这些行向 Yacc 指出算符 + 和 - 具有优先权且是左结合的，而且运算符 * 是左结合且有比 + 和 - 更高的优先权 (因为在说明中，它是列在这些算符之后)。在 Yacc 中，其他可能的算符说明是 %right 和 %nonassoc ("nonassoc" 意味着重复的算符不允许出现在相同的层次上)。

程序清单 5-5 带有二义性文法和算符的优先权及结合性规则的简单计算器的 Yacc 说明

```

%{
#include <stdio.h>
#include <ctype.h>
%}

%token NUMBER

%left '+' '-'
%left '*'

```

```

%%

command : exp      { printf("%d\n", $1); }
        ;

exp      : NUMBER      { $$ = $1; }
        | exp '+' exp  { $$ = $1 + $3; }
        | exp '-' exp  { $$ = $1 - $3; }
        | exp '*' exp  { $$ = $1 * $3; }
        | '(' exp ')'  { $$ = $2; }
        ;

%%

/* auxiliary procedure declarations as in Figure 5.10 */

```

5.5.4 描述Yacc分析程序的执行

除了在y.output文件中显示分析表的详细选项之外，Yacc生成的分析程序也可能打印出它的执行过程，这其中包括了分析栈的一个描述以及与本章早先给出的描述类似的分析程序动作。这是通过用符号定义的YYDEBUG编译y.tab.c(例如通过使用-DYYDEBUG编译选项)以及在需要描述信息的地方将Yacc整型变量yydebug设置为1。例如假设表达式2+3是输入，请将下面的行添加到程序清单5-1的main过程的开头

```

extern int yydebug;
yydebug = 1;

```

将会使得分析程序产生一个与程序清单5-6相似的输出。我们希望读者利用表5-11的分析表手工构造出分析程序的动作描述并将它与这个输出作一对比。

程序清单5-6 假设有输入2+3，利用yydebug为由程序清单5-1生成的Yacc分析程序描绘输出

```

Starting parse
Entering state 0
Input: 2+3
Next token is NUMBER
Shifting token NUMBER, Entering state 5
Reducing via rule 7, NUMBER -> factor
state stack now 0
Entering state 4
Reducing via rule 6, factor -> term
state stack now 0
Entering state 3
Next token is '+'
Reducing via rule 4, term -> exp
state stack now 0
Entering state 2
Next token is '+'
Shifting token '+', Entering state 7
Next token is NUMBER
Shifting token NUMBER, Entering state 5
Reducing via rule 7, NUMBER -> factor
state stack now 0 2 7
Entering state 4
Reducing via rule 6, factor -> term

```

```

state stack now 0 2 7
Entering state 11
Now at end of input.
Reducing via rule 2, exp '+' term -> exp
state stack now 0
Entering state 2
Now at end of input.
Reducing via rule 1, exp -> command
5
state stack now 0
Entering state 1
Now at end of input.

```

5.5.5 Yacc中的任意值类型

程序清单 5-1 使用与文法规则中的每个文法符号相关的 Yacc 伪变量指出计算器的动作。例如，用写出 `$$ = $1 + $2` 将一个表达式的值设置为它的两个子表达式值之和（在文法规则 `exp → exp + term` 的右边的位置 1 和位置 3）。由于这些值的 Yacc 缺省类型总是整型，所以只要处理的值是整型就可以了。但是若要用浮点值计算一个计算器，那就不合适了。在这种情况下，必须在说明文件中对 Yacc 伪变量的值类型进行重定义。这个数据类型总是由 C 预处理器符号 `YYSTYPE` 在 Yacc 中定义。重定义这个符号会恰当地改变 Yacc 值栈的类型。因此若需要一个计算浮点值的计算器，就必须在 Yacc 说明文件的定义部分的括号 `{...}` 中添加行

```
#define YYSTYPE double
```

在更复杂的情况下，不同的文法规则就有可能需要不同的值。例如，假设希望将算符挑选的识别与用那些算符计算的规则分隔开，如在规则

```

exp    exp addop term | term
addop  + | -

```

（它们实际上是表达式文法的原始规则，我们把它们改变了以直接识别程序清单 5-1 中的 `exp` 规则的算符）。现在 `addop` 必须返回算符（一个字符），但 `exp` 必须返回计算的值（即一个 `double`），但这两个数据类型不同。我们所需要做的是将 `YYSTYPE` 定义为 `double` 与 `char` 的联合。有两种方法可以办到。其一是在 Yacc 说明中利用 Yacc 的 `%union` 声明来直接声明一个联合：

```
%union { double val;
         char op; }
```

现在 Yacc 需要被告知每个非终结符的返回类型，这是利用定义部分中的 `%type` 指示来实现的：

```
%type <val> exp term factor
%type <op> addop mulop
```

请注意，在 Yacc 的 `%type` 声明中，联合域的名称是怎样被尖括号括起来的。接着将程序清单 5-1 的 Yacc 说明修改成按下开始（我们将详细的写法留在练习中）：

```

...
%token NUMBER

%union { double val;
        char op; }

%type <val> exp term factor NUMBER
%type <op> addop mulop

```

```

%%
comand : exp          { printf ( " %d\n " , $1 ) ; }
      ;
exp    : exp op term { swithc ( $2 ) {
                        case '+': $$ = $1 + $3; break;
                        case '-': $$ = $1 - $3; break;
                        }
      }
      | term { $$ = $1; }
      ;
op    : '+' { $$ '+'; }
      | '-' { $$ '-'; }
      ;

```

第2种方法是在另一个包含文件(例如,一个头文件)中定义一个新的数据类型之后再将YYSTYPE定义为这个类型。接着必须在相关的动作代码中手工地构造出恰当的值来。下一节中的TINY分析程序是它的一个示例。

5.5.6 Yacc中嵌入的动作

在分析时,有时需要在完整地识别一个文法规则之前先执行某个代码。例如考虑简单声明的情况:

```

decl    typevar-list
type    int | float
var-list var-list , id | id

```

当识别var-list时,用当前类型(整型和浮点)将标签添加于每个变量标识符上。可按如下方法在Yacc中完成:

```

decl  : type { current_type = $1 ; }
      var_list
      ;
type  : INT { $$ = INT_TYPE ; }
      | FLOAT { $$ = FLOAT_TYPE ; }
      ;
var_list : var_list ',' ID
          { setType (tokenString, current_type);}
          | ID
          { setType (tokenString, current_type);}
          ;

```

请注意,在decl规则中识别变量之前,一个嵌入的动作如何设置current_type。读者将在后面的章节中看到其他有关嵌入动作的示例。

Yacc将一个嵌入动作

```
A : B { /* embedded action */ } C ;
```

解释为与一个新的占位符非终结符相等价,且与在被归约时,执行嵌入动作的非终结符的 ϵ -产生式等价:

```

A : B E C ;
E : { /* embedded action */ } ;

```

最后，在表5-12中小结了Yacc的定义机制以及已讨论过的一些内置名称。

表5-12 Yacc内置名称和定义机制

Yacc的内置名称	含义 / 用处
<code>y.tab.c</code>	Yacc输出文件名称
<code>y.tab.h</code>	Yacc生成的头文件，包含了记号定义
<code>yyparse</code>	Yacc分析例程
<code>yyval</code>	栈中当前记号的值
<code>yyerror</code>	由Yacc使用的用户定义的错误信息打印机
<code>error</code>	Yacc错误伪记号
<code>yyerrok</code>	在错误之后重置分析程序的过程
<code>yychar</code>	包括导致错误的先行记号
<code>YYSTYPE</code>	定义分析栈的值类型的预处理器符号
<code>yydebug</code>	变量，当由用户设置为1时则导致生成有关分析动作的运行信息

Yacc的定义机制	含义 / 用处
<code>%token</code>	定义记号预处理器符号
<code>%start</code>	定义开始非终结符号
<code>%union</code>	定义和 <code>YYSTYPE</code> ，允许分析程序栈上的不同类型的值
<code>%type</code>	定义由一个符号返回的和类型
<code>%left %right %nonassoc</code>	定义算符的结合性和优先权(由位置)

5.6 使用Yacc生成TINY分析程序

TINY的语法已在3.7节中给出，且在4.4节中也已给出了一个手写的分析程序；我们希望读者能掌握这些内容。这里将描述 Yacc说明文件`tiny.y`以及对全程定义`globals.h`的修改(我们已采用了一些方法将对其他文件的改变定为最小，这也将讲到)。整个`tiny.y`文件都列在了附录B的第4000行到第4162行中。

首先讨论TINY的Yacc说明中的定义部分。稍后将谈到标志 `YYPARSER`(第4007行)。表示了Yacc在程序中的任何地方都需要的信息有4个`#include`文件(第4009行到第4012行)。定义部分有4个其他声明。第1个(第4014行)是`YYSTYPE`的定义，它定义了通过使Yacc分析过程为指向节点结构的指针返回的值(`TreeNode`本身被定义在`globals.h`中)，这样就允许了Yacc分析程序构造出一个语法树。第2个是全程`savedName`变量的定义，它被用作暂时储存要被插入到还没构造出的树节点中的标识字符串，而此时已能在输入中看到这些串了(在TINY中只有在赋值中才需要)。变量`savedLineNo`也是被用作相同目的，所以那个恰当的源代码行数也将与标识符关联。最后，`savedTree`被用来暂时储存由`yyparse`过程产生的语法树(`yyparse`本身可以仅返回一个整型标志)。

下面讨论一下与TINY的每个文法规则相结合的动作(这些规则与第3章的程序清单3-1中所给出的BNF略有不同)。在绝大多数情况下，这些动作表示与该点上的分析树相对应的语法树的构造。特别地，需要从`util`包调用到`newStmtNode`和`newExpNode`来分配新的节点(这些已在4.4节中讲述过了)，而且也需要指派新树节点的合适的子节点。例如，与 TINY的`write_stmt`(第4082ff行)相对应的动作如下所示：


```

write_stmt : WRITE exp
            { $$ = newStmtNode (WriteK);
              $$ ->child [0] = $2;
            }
;

```

第1个指令调用newStmtNode并指派返回的值为write_stmt的值。接着exp (Yacc伪变量\$2是指向将要被打印的表达式树节点的指针)前面构造的值为write语句的树节点的第1个孩子。其他的语句和表达式的动作代码十分类似。

program、stmt_seq和assign_stmt的动作处理与每个这些构造相关的小问题。在program的文法规则中,相关的动作(第4029行)是

```
{savedTree = $1;}
```

它将stmt_seq构造的树赋给了静态变量savedTree。由于它使得语法树可由parse过程之后返回,所以这是必要的。

在assign_stmt的情况中,我们早已指出需要储存作为赋值目标的变量的标识字符串,这样当构造节点时(以及为了便于今后的描绘,还编制了它的行号)它就是恰当的了。通过使用savedName和saveLineNo静态变量(第4067行)可以做到这一点:

```

assign_stmt : ID { savedName = copyString ( tokenString ) ;
                  savedLineNo = lineno ; }
            ASSIGN exp
            { $$ = newStmtNode ( AssignK );
              $$ ->child [ 0 ] = $4 ;
              $$ ->attr.name = savedName ;
              $$ ->lineno = saveLineNo ;
            }
;

```

由于作为被匹配的新记号,tokenString和lineno的值都被扫描程序改变了,所以标识字符串和行号必须作为一个在ASSIGN记号识别之前的一个嵌入动作储存起来。但是只有在识别出exp之后才能完全构造出赋值的新节点,因此就需要savedName和saveLineNo。(实用程序过程copyString的使用确保了这些串没有共享存储。读者还需注意将exp的值认为是\$4。这是因为Yacc认为嵌入动作在文法规则的右边是一个额外的位置——参见上一节的讨论)。

在stmt_seq(第4031行到第4039行)的情况中,它的问题是:属指针(而不是孩子指针)将语句在TINY语法树中排在一起。因为人们将语法序列的规则写成左递归的,这也就要求为了在末尾附上当前的语句,代码应找出早先为左子集构造的属列表。这样做的效率并不高而且我们还可以通过将规则重写为右递归来避免它,但是这个解决方法也有它自身的问题:只要处理语句序列,其中的分析栈就会变得很大。

最后,Yacc说明的辅助过程部分(第4144行到第4162行)包括了3个过程——yyerror、yylex和parse——的定义。parse过程是由主程序调用,它将调用Yacc定义的分析过程yyparse并且返回保存的语法树。之所以需要yylex过程是因为Yacc假设这是扫描程序过程的名称,而它又在外部被定义为getToken。写出了这个定义就使得Yacc生成的分析程序可在对别的代码文件只作出最小的改变的情况下就可与TINY编译程序一起工作。有人可能希望对扫描程序作出恰当的改变并省掉这个定义,特别是在使用扫描程序的Lex版本时。在出现错误时,yyerror过程由Yacc调用:它将一定的有用信息(如行号)打印到列表文件上。它使用的是

Yacc的内置变量`yychar`，`yychar`包含了引起错误的记号的记号号码。

我们还需要描述对 TINY 分析程序中的其他文件的改变，由于使用了 Yacc 来产生分析程序，所以这些改变也是必要的。正如前面已指出的，我们的目标是使这些改变成为最小的，而且将所有的改变都限制为 `globals.h` 文件的。修改过的文件列在了附录 B 的第 4200 行到第 4320 行中。其基本问题是由 Yacc 生成的包含了记号定义的头文件必须被包括在大多数的其他代码文件中，但它又不能直接被包括在 Yacc 生成的分析程序中，因为这样做会重复内置的定义。对上面问题的解决办法是用一个标志 `YYPARSER` (第 4007 行) 的定义作为 Yacc 说明部分的开头，而 `YYPARSER` 位于 Yacc 分析程序中且指出 C 编译程序何时处于分析程序之中。我们使用那个标志 (第 4226 行到第 4236 行) 有选择地将 Yacc 生成的头文件 `y.tab.h` 包括在 `globals.h` 中。

第 2 个问题出现在 `ENDFILE` 记号中，此时扫描程序要指出输入文件的结尾。Yacc 假设这个记号总是存在着值 0，因此也就提供了这个记号的直接定义 (第 4234 行) 并且将它包括在由 `YYPARSER` 控制的有选择性的编译部分之中，这是因为 Yacc 内部并不需要它。

因为所有的 Yacc 记号都有整型值，所以 `globals.h` 文件最后的改变是将 `TokenType` 重定义为 `int` 的一个同义字 (第 4252 行)。这样就避免了无必要地替代前面所列的其他文件中的类型 `TokenType`。

5.7 自底向上分析程序中的错误校正

5.7.1 自底向上分析中的错误检测

当在分析表中检测到一个空 (或错误) 项时，自底向上的分析程序将检测错误。尽可能快地检测到错误显然是有意义的，这样的错误信息可以更有意义且是确定的。因此，分析表应有尽可能多的空项。

不幸的是，这个目标与一个同等重要的目标相冲突，这就是缩小分析表的大小。我们早已看到 (在表 5-11 中) Yacc 尽可能多地用缺省归约来填充表项，所以在声明错误之前，大量的归约将占据分析栈。这样就会使错误的准确来源不清晰，而且会导致没有意义的错误信息。

自底向上的分析的另一个特征是所使用的特定算法的能力会影响到分析程序是否可早些检测出错误的能力。例如一个 LR(1) 分析程序能够比 LALR(1) 分析程序或 SLR(1) 分析程序更早地检测出错误来；而在这方面，LALR(1) 分析程序和 SLR(1) 分析程序又比 LR(0) 分析程序能力强一些。例如，对比 LR(0) 分析表 (表 5-4) 和 LR(1) 分析表 (表 5-10) 中的相同文法。假设存在着错误的输入串 (`a $`，表 5-10 中的 LR(1) 分析表就会将 (`a` 移进到栈中的状态 6。由于在状态 6 中没有项是位于 `$` 之下，所以就报告了一个错误。相反地，LR(0) 算法 (以及 SLR(0) 算法) 在发现缺少右括号之前先用 `A a` 归约。类似地，假设存在着错误串 `a) $`，则一般的 LR(1) 分析程序会移进 `a`，接着再声明来自右括号上的状态 3 的错误；而 LR(0) 分析程序在声明错误之前先用 `A a` 归约。当然，任何的自底向上的分析程序总是可能会在若干个“错误的”归约之后最终报告出错误来。这些分析程序都不会移进错误的记号。

5.7.2 应急方式错误校正

在自底向上的分析中，通过明智地从分析栈或输入或以上这两者中删除符号有可能会适度地在自底向上的分析程序中得到较好的错误校正。与 LL(1) 分析程序相似，它有可能完成 3 种动作：

- 1) 从栈中弹出一个状态。
- 2) 在看到可重新开始分析的记号之后，就从输入中成功地弹出记号。
- 3) 将一个新的状态压入到栈中。

当发生错误时，用来选择以上哪个动作的特别有效的方法如下：

- 1) 在发现一个带有非空的Goto项的状态之后从分析栈中弹出状态。
- 2) 若在当前的输入记号上有一个来自Goto状态的合法动作，则将这个状态压入到栈中，并重新开始分析。如果存在着若干个这样的状态，则选择移进而不是归约。在归约动作中，则选择其结合的非终结符为最不一般的那一个。

3) 若在当前输入记号上没有来自Goto状态的合法动作，推进输入直到有一个合法的动作或到达了输入的末尾。

这些规则能够强迫一个构造的识别，当错误发生时这个构造位于正被识别的处理中，并由此立即重新开始分析。使用这些或类似规则的错误校正可被称作应急方式 (panic mode) 错误校正，这是因为它与在4.5节中描述的自顶向下的应急方式类似。

不幸的是，由于步骤2将新的状态压入栈中，所以这些规则将会导致一个无穷循环。此时可有若干个可能的解决办法。其一是在步骤2上坚持来自一个Goto状态的移进动作；但是这可能会有太大的限制。另一个办法是：若下一个合法的移动是归约，则设置一个引起分析程序跟踪在下面归约中的状态序列的标志；若相同的状态重现时，则直到在错误发生时将初始的状态删去之后再弹出栈状态，并且再次从步骤1开始。若在任何时候都发生了移进，则分析程序重新设置标志并开始正常的分析。

例5.19 在一个简单的算术表达式文法中(它的Yacc分析表在表5-11中给出)，现在考虑错误的输入(2+*)。在看到*之前，分析一直是正常进行。此时，应急方式会导致在分析栈中发生以下的动作：

分 析 栈	输 入	动 作
...
\$0 (6 E 10+7	*) \$	错误： 压入T, goto 11
\$0 (6 E 10+7 T 11	*) \$	移进9
\$0 (6 E 10+7 T 11 * 9) \$	错误： 压入F, goto 13
\$0 (6 E 10+7 T 11 * 9 F 13) \$	用T T * F 归约
...

在第1个错误中，分析程序位于状态7中，它有合法的Goto状态11和4。由于状态11在下一个输入记号*上存在着一个移进，而这正是Goto倾向的，所以就将记号移进。此时分析程序位于状态9中，在输入中它有一个右括号。而这又是一个错误。在状态9中，有一个单个的Goto项(到状态11)，且状态11在)上也确有一个合法的动作(虽然是一个归约)。分析接着就正常地继续它的结论。

5.7.3 Yacc中的错误校正

不用应急方式，我们可以使用称作错误产生式 (error production) 的方法。错误产生式就是

一个包括了伪记号 **error** 作为它的右边的唯一符号的产生式。错误产生式标志着一个上下文，直到看到恰当的同步记号时，在其中的错误记号才被删除，而此时又可重新开始分析。错误产生式可有效地允许程序员用手写标记出其 Goto 项将被用作错误校正的非终结符。

错误产生式是在 Yacc 中用于错误恢复的主要方法。当发生错误时，Yacc 分析程序的行为以及它处理错误产生式的行为如下所示。

1) 当分析程序在分析中检测到错误时(即，它遇到分析表中的一个空项)，它会从分析栈中弹出状态直至到达一个其中的 **error** 伪记号是合法的先行的状态。其结果是将输入丢弃到错误的左边，并将输入看作是包括了 **error** 伪记号。如果没有错误伪记号，则 **error** 永远也不会是移进的合法先行，而且分析栈也将为空，它使分析在第 1 个错误处中断(这是由程序清单 5-1 的输入 Yacc 生成的分析程序行为)。

2) 一旦分析程序找到了栈上的一个状态，在该状态中的 **error** 就是一个合法的先行，它以正常的风格继续移进和归约。其结果是好像在输入中看到了 **error**，它的后面是初始先行(也就是导致错误的先行)。如若需要，Yacc 宏 **yyclearin** 可被用来丢弃掉引起错误的记号，并将它随后的记号作为下一个先行(在 **error** 之后)。

3) 如果分析程序在一个错误发生之后发现了更多的错误，则直到将 3 个成功的记号合法地移进到分析栈中之后为止，引起错误的输入记号才会被无声地丢弃掉。此时认为分析程序是位于一个“错误状态”之中。将这个行为设计用来避免由相同错误引起的错误信息级联。但这样就会导致在分析程序退出错误状态之前丢掉大量的输入(同在应急方式中一样)。编译程序的编写者可以利用 Yacc 宏 **yyerrorok** 将分析程序从错误状态中删除掉以不考虑这个行为，所以在没有新的错误校正的情况下就不会丢掉更多的输入了。

根据程序清单 5-1 中的 Yacc 输入，我们再描述这个行为的几个简单示例。

例 5.20 考虑程序清单 5-1 中 **command** 规则的以下替换：

```
command : exp      { printf ("%d\n", $1); }
        | error    { yyerror ("incorrect expression"); }
        ;
```

再考虑错误的输入 **2++3**。这个串的分析在到达第 2 个 **+** 之前一直是正常的。此时的分析是由以下的分析栈和输入给出的(虽然错误产生式的加法实际上将会导致分析表的少许变化，但我们还是用表 5-11 来描述它)：

PARSING STACK	INPUT
\$0 exp 2 + 7	+3\$

现在分析程序输入错误“状态”(生成一个诸如“语法错误”的错误信息)，从栈开始弹出状态直到发现状态 0。此时，**command** 的错误产生式提供 **error** 为合法的先行，而且将它移进到分析栈中，并立即归约到 **command** 上，它执行了相结合的动作(该动作打印出信息“不正确的表达式”)。最后的状况如下所示：

PARSING STACK	INPUT
\$0 command 1	+3\$

此时唯一的合法先行是输入的末尾(在这里由 **\$** 指出，它与由 **yylex** 返回的 0 相对应)，而且分析程序在退出之前(但仍在“错误状态”中)将删除输入记号 **+3** 的剩余部分。因此，除了现在能够提供自己的错误信息之外，错误产生式的加法具有了同程序清单 5-1 中版本相同的效果。

一个比这个更好的错误机制允许用户在错误输入之后重新输入行。此时需要一个同步的记

号，而且行标记的末尾是唯一敏感的。因此，扫描程序必须经过修改而返回新行字符（而不是0），经过这个修改可以写出(参见练习5.32)：

```
command : exp ' \n ' { printf ( ' %d\n' , $1); exit (0); }
        | error ' \n '
          { yyerrok ;
            printf ( "reenter expression : " ); }
        command
    ;
```

这个代码的结果是：当发生错误且当它执行由 **yyerrok** 表示的动作和 **printf** 语句时，分析程序将跳过所有的记号而到达一个新行。接着它将试图识别出另一个 *command*。这里需要一个向 **yyerrok** 的调用来在看到新行之后删除“错误状态”；这是因为如若不然，则当新行的右边发生新的错误时，Yacc在直到发现3个正确的记号序列之后才会无声地删除掉记号。

例5.21 若按以下的表示将一个错误产生式添加到程序清单 5-1的Yacc定义中，那么会出现怎样的结果呢：

```
factor      : NUMBER          { $$ = $1; }
            | ' ( ' exp ' ) ' { $$ = $2; }
            | error { $$ = 0; }
            ;
```

首先考虑与前例相同的错误输入 $2++3$ 。(尽管加法错误产生式使得表发生了略微的改变，但我们仍然使用表5-11。)分析程序与上述相同，将会到达以下的输入：

PARSING STACK	INPUT
\$0 exp 2 + 7	+3\$

现在 *factor* 的错误产生式将提供 **error** 为状态7中的一个合法先行，而且将 **error** 立即移进到栈中，并归约到引起返回值0的 *factor* 上。现在分析程序已到达了以下的点：

PARSING STACK	INPUT
\$0 exp 2 + 7 factor 4	+3\$

这是一个正常的情况，而且分析程序将继续正常到执行的结束。其结果是将输入解释为 $2+0+3$ ，将0放在两个+符号之间是因为此处是 **error** 伪记号插入的地方，而且通过错误产生式的动作，**error** 被看作是带有0值的一个因子等价。

现在考虑错误的输入 $2\ 3$ (即缺少了运算符的两个数字)。分析程序到达了位置

PARSING STACK	INPUT
\$0 exp 2	3\$

此时(若 *command* 的规则并未改变)即使数字并不是 *command* 的合法后随符号，分析程序也将(错误地)用规则 *command exp* 归约(并打印出值2)。接着，分析程序到达位置

PARSING STACK	INPUT
\$0 command 1	3\$

现在删除了一个错误，且在揭示状态0的同时从分析栈中弹出状态1。此点 *factor* 的错误产生式允许为 **error** 成为一个来自状态0的合法先行，而且 **error** 被移进到分析栈中，并导致打印归约的另一个级联以及值0(由错误产生式返回的值)。现在分析程序又回到了分析的相同位置，而且数字3仍在输入中！幸运地是，分析程序早已在“错误状态”中并不再移进 **error** 了，但

是却扔掉了数字3，它揭示了状态1的正确先行，因此分析程序仍然存在[⊖]。其结果是分析程序按如下所示打印(在第1行中重复用户的输入)：

```
> 2 3
2
syntax error
incorrect expression
0
```

该行为大致给出了Yacc中良好的错误恢复的困难(参见练习中的更多练习)。

5.7.4 TINY中的错误校正

附录B中的Yacc说明文件tiny.y包括了两个错误产生式，一个是stmt的(第4047行)，另一个是factor的(第4139行)，与之相关的动作将返回空的语法树。除了它不试图在错误中建立重要的语法树之外，这些错误产生式提供一个与上一章中的递归下降TINY分析程序中相似的错误处理级别。这些错误产生式还不能为分析的重新开始提供同步，所以在多重错误中，会跳过多个记号。

练习

5.1 考虑以下的文法：

$$\begin{aligned} E & \rightarrow (L) \mid a \\ L & \rightarrow L, E \mid E \end{aligned}$$

- 为这个文法构造LR(0)项目的DFA。
- 构造SLR(1)分析表。
- 显示分析栈和输入串

((a), a, (a,a))

的SLR(1)分析程序的动作。

- 这个文法是不是LR(0)文法？若不是，请描述出LR(0)冲突。如果是，则构造LR(0)分析表，并描述一个分析如何可以与SLR(1)分析不同。

5.2 考虑上面练习中的文法

- 为这个文法构造LR(1)项目的DFA。
- 构造一般的LR(1)分析表。
- 为这个文法构造LALR(1)项目的DFA。
- 构造LALR(1)分析表。
- 描述任何可能出现在一般的LR(1)分析程序的动作和LALR(1)分析程序的动作之间的区别。

5.3 考虑以下的文法：

$$A \rightarrow A(A) \mid \varepsilon$$

- 为这个文法构造LR(0)项目的DFA。
- 构造SLR(1)分析表。

[⊖] Yacc的一些版本在删除任何输入之前先再次弹出分析栈。这样会导致更复杂的行为。参见练习。

c. 显示分析栈和输入串

((()())

的SLR(1)分析程序的动作。

d. 这个文法是不是 LR(0)文法？如果不是，请描述出 LR(0)冲突。如果是，则构造 LR(0)分析表，并描述一个分析如何与SLR(1)分析相区别。

5.4 考虑上一个练习的文法

a. 为这个文法构造LR(1)项目的DFA。

b. 构造一般的LR(1)分析表。

c. 为这个文法构造LALR(1)项目的DFA。

d. 构造LALR(1)分析表。

e. 描述任何可能出现在一般的 LR(1)分析程序的动作和LALR(1)分析程序的动作之间的区别。

5.5 考虑简化了的语句序列的以下文法：

$$\begin{aligned} \text{stmt-sequence} & \quad \text{stmt-sequence} ; \text{stmt} \mid \text{stmt} \\ \text{stmt} & \quad \text{S} \end{aligned}$$

a. 为这个文法构造LR(0)项目的DFA。

b. 构造SLR(1)分析表。

c. 显示分析栈和输入串

S ; S ; S

的SLR(1)分析程序的动作。

d. 这个文法是不是 LR(0)文法？如果不是，请描述出 LR(0)冲突。如果是，则构造 LR(0)分析表，并描述一个分析如何与一个SLR(1)分析相区别。

5.6 考虑上一个练习的文法

a. 为这个文法构造LR(0)项目的DFA。

b. 构造一般的LR(1)分析表。

c. 为这个文法构造LALR(1)项目的DFA。

d. 构造LALR(1)分析表。

e. 描述任何可能出现在一般的 LR(1)分析程序的动作和LALR(1)分析程序的动作之间的区别。

5.7 考虑以下的文法：

$$\begin{aligned} E & \quad (L) \mid a \\ L & \quad E L \mid E \end{aligned}$$

a. 为这个文法构造LR(0)项目的DFA。

b. 构造SLR(1)分析表。

c. 显示分析栈和输入串

((a)a(a a))

的SLR(1)分析程序的动作。

d. 通过在LR(0)项目的DFA中传送先行来构造LALR(1)项目的DFA。

e. 构造LALR(1)分析表。

5.8 考虑以下的文法


```

declaration    type var-list
type           int | float
var-list       identifier, var-list | identifier

```

- a. 将它用一个更适于自底向上分析的格式重写一次。
- b. 为重写的文法构造 LR(0) 项目的 DFA。
- c. 为重写的文法构造 SLR(1) 分析表。
- d. 利用 c 部分的表为输入串 `int x, y,` 显示分析栈和 SLR(1) 分析程序的动作。
- e. 通过在 b 部分中的 LR(0) 项目的 DFA 中传送先行来构造 LALR(1) 项目的 DFA。
- f. 为重写的文法构造 LALR(1) 分析表。

5.9 为通过在 LR(0) 项目的 DFA 中传送先行而构造 LALR(1) 项目的 DFA 写出算法的正式描述(在 5.4.3 节中已经非正式地描述过这个算法了)。

5.10 本章显示的所有分析栈都包括了状态数和文法符号(为了清楚起见), 但是分析栈仅需要储存状态数即可——无需将记号和非终结符存放在栈中。若只将状态数保存在栈中, 请描述出 SLR(1) 分析算法。

5.11 a. 说明以下的文法不是 LR(1) 文法:

$$A \rightarrow a A a \mid \varepsilon$$

b. 这个文法有二义性吗? 为什么?

5.12 说明以下的文法是 LR(1) 文法但不是 LALR(1) 文法:

$$\begin{aligned}
 S &\rightarrow a A d \mid b B d \mid a B e \mid b A e \\
 A &\rightarrow c \\
 B &\rightarrow c
 \end{aligned}$$

5.13 说明不是 LALR(1) 文法的 LR(1) 文法只能有归约-归约冲突。

5.14 说明当且仅当一个右句子格式的前缀不会扩展到句柄之外时, 它才能是一个变量前缀。

5.15 有没有一个 SLR(1) 文法不是 LALR(1) 文法的? 为什么?

5.16 说明如果不能最终地将下一个输入记号移进, 则一般的 LR(1) 分析程序在声明一个错误之前不会归约。

5.17 SLR(1) 分析程序会不会在声明错误之前实现的归约比 LALR(1) 分析程序实现的少? 请解释原因。

5.18 以下的有二义性的文法生成了与练习 5.3 中的文法相同的串(即: 嵌套的括号的所有串):

$$A \rightarrow AA \mid (A) \mid \varepsilon$$

由 Yacc 生成的分析程序将使用这个文法识别所有的合法串吗? 为什么?

5.19 假设有在其中有两个可能的归约(在不同的先行上)的状态, Yacc 将选择其中的一个归约作为它的缺省动作。请描述出 Yacc 在作出这个选择时所使用的规则(提示: 使用练习 5.16 中的文法作为一个测试情况)。

5.20 假设从程序清单 5-5 的 Yacc 说明中删除了算符的结合性和优先权的指定(因此就剩下一个有二义性的文法)。请描述出 Yacc 缺省的消除二义性的规则产生了怎样的结合性和优先权。

5.21 同例 5.21 中脚注所描述的一样, 有一些 Yacc 分析程序在丢弃任何位于“错误状态”

之中的输入之前会再次弹出分析栈。假设错误输入是 2 3，则请描述出例5.21中的Yacc说明对这样的分析程序的行为。

- 5.22 a. 利用分析表5-11和串(*2描绘出如5.7.1节所描述的应急方式错误校正机制。
b. 对a部分中的行为建议一个改善办法。
- 5.23 假设程序清单5-1的Yacc计算器说明中的command规则是由一个list开始的非终结符替换的：

```
list      :      list ' \n' { exit (0); }
          |      list exp ' \n' { printf ("%d\n", $2); }
          |      list error ' \n' { yyerrok }
          ;
```

且这个图中的yylex过程将行

```
if (c == ' \n ') return 0;
```

删除掉了。

- a. 解释简单计算器的这个版本的行为与程序清单5-1中版本的行为之间的差别。
b. 解释这个规则最后一行的yyerrok的原因。给出一个显示若它不在这里的情况的示例。
- 5.24 a. 假设程序清单5-1中的Yacc计算器说明中的command的规则被以下的规则
- ```
command : exp error { printf (" %d\n:", $1); }
```
- 替代了，则若输入为2 3，请准确地解释出Yacc分析程序的行为。  
b. 假设程序清单5-1中的Yacc计算器的说明的command的规则被以下的规则
- ```
command : error exp { printf (" %d\n", $2); }
```
- 替代了，则若输入为2 3，请准确地解释出Yacc分析程序的行为。
- 5.25 假设例5.21中的Yacc错误产生式被以下的规则

```
factor    : NUMBER          { $$ = $1; }
          | ' (' exp ')'     { $$ = $2; }
          | error { yyerrok; $$ = 0; }
          ;
```

替换了。

- a. 解释在错误的输入2++3中，Yacc分析程序中的行为。
b. 解释在错误的输入2 3中，Yacc分析程序中的行为。
- 5.26 利用4.5.3节中的测试程序对比由Yacc生成的TINY分析程序的错误校正和第4章中的递归下降分析程序。解释二者行为中的不同之处。

编程练习

- 5.27 重写程序清单5-1中的Yacc说明以使用以下的文法规则(而不是scanf)计算出一个数的值(以及，因此舍弃掉NUMBER记号)：

```
number    number digit | digit
digit     0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

- 5.28 将以下添加到程序清单5-1中的Yacc整型计算说明中(确保它们具有正确的结合性和优先级)：
- a. 带有符号/的除法。

- b. 带有符号%的整型模。
- c. 带有符号^的整型求幂(警告：这个算符的优先权比乘法的高，且为右结合)。
- d. 带有符号-的一目减。

- 5.29 将程序清单 5-1 中的 Yacc 计算器说明重新改写以使计算器会接受浮点数 (并执行浮点计算)。
- 5.30 重写程序清单 5-1 中的 Yacc 计算说明以使其可区别浮点值与整型值，而不是仅仅将任何东西都作为整型数或浮点数来计算。(提示：“值”现在是一个带有指出它是整型还是浮点的标志)。
- 5.31 a. 将程序清单 5-1 中的 Yacc 计算器说明重写以使它根据 3.3.2 节中的说明会返回一个语法树。
b. 写出一个函数，由 a 部分中的代码生成的语法树作为参数并返回由遍历语法树计算出的值。
- 5.32 为例 5.20 中的计算器程序建议的简单错误校正技术有一个缺点：它在许多错误之后可导致栈满溢。请改写它以解决这个问题。
- 5.33 重写程序清单 5-1 中的 Yacc 计算器说明以添加以下的有用的错误信息：
由串 (2 +3 生成的“丢失右括号”
由串 2+3) 生成的“丢失左括号”
由串 2 3 生成的“丢失算符”
由串 (2+) 生成的“丢失操作数”
- 5.34 以下的文法表示在一个类似于 LISP 的前缀表示法中的简单的算术表达式：

$$\begin{aligned} \text{lexp} & \quad \text{number} \mid (\text{op lexp-seq}) \\ \text{op} & \quad + \mid - \mid * \\ \text{lexp-seq} & \quad \text{lexp-seq lexp} \mid \text{lexp} \end{aligned}$$

例如，表达式 $(* (- 2) 3 4)$ 具有值 -24。为一个在这个语法中计算并打印表达式的值的程序写出一个 Yacc 说明(提示：这要求重写文法以及使用将算符分析为一个 lexp-seq 的机制)。

- 5.35 以下的文法代表了第 2 章曾讨论过的正则表达式：

$$\begin{aligned} \text{rexp} & \quad \text{rexp} " \mid " \text{rexp} \\ & \quad \mid \text{rexp} \text{rexp} \\ & \quad \mid \text{rexp} " * " \\ & \quad \mid " (\text{rexp}) " \\ & \quad \mid \text{letter} \end{aligned}$$

- a. 写出一个表达该运算的正确结合性和优先权的大致的 Yacc 说明(即：没有动作)。
 - b. 将 a 部分中的说明扩展到包括产生一个“正则表达式编译程序”的所有动作和辅助过程。也就是说，一个在编译时将正则表达式作为 C 程序的输入和输出的程序为第一次遇到的匹配正则表达式的子集搜索一个输入串(提示：可将表或两维数组代表状态和相关的 NFA 的转换。接着可利用一个列来储存状态来模拟 NFA。只有表才需被 Yacc 动作生成：剩余的代码将总是相同的，参见第 2 章)。
- 5.36 将 TINY 的 Yacc 说明(附录 B 的第 4000 行到第 4162 行)用以下之一的方法在更简洁的格式中重写一次：

- a. 通过为表达式使用有二义性的文法（和 Yacc 对于优先权和结合性带有消除二义性的规则）
- b. 通过将算符的识别成一个单一的规则，如在

$$\begin{aligned} \text{exp} & \quad \text{exp op term} \mid \dots \\ \text{op} & \quad + \mid - \mid \dots \end{aligned}$$

以及通过使用 Yacc 的 %union 声明 (允许 op 返回算符，但 exp 和其他的非终结符却返回指向树节点的指针)。确保你的分析程序产生了与前面相同的语法树。

- 5.37 将比较算符 <= (小于或等于)、> (大于)、>= (大于或等到于)，和 <> (不等到于) 添加到 TINY 分析程序中的 Yacc 说明 (这将要求添加这些记号并改变扫描程序，却不要求改变语法树)。
- 5.38 将布尔算符 and、or 和 not 添加到 TINY 分析程序的 Yacc 说明中。假设它们具有练习 3.5 中描述的特性以及比所有的算术算符都低的优先权，确保任何表达式都可为布尔或整型。
- 5.39 重写 TINY 分析程序的 Yacc 说明以使其可改进它的错误校正。

注意与参考

一般的 LR 是由 Knuth [1965] 发明的，但人们直到 SLR 和 LALR 技术被 DeRemer [1969, 1971] 开发出来之前一直都是认为它是不实际的。我们已重复了 LR(1) 分析程序对于实际应用非常复杂的惯例。实际上，利用比 LALR(1) 分析的技术更精致的状态结合技术可建立实际的 LR(1) 分析程序 [Pager, 1977]。然而却很少用到附加的能力。在 Aho 和 Ullman [1972] 中可找到有关 LR 分析技术理论的完整研究。

Yacc 是在 20 世纪 70 年代由 Steve Johnson 在 AT&T 的贝尔实验室开发并包括在大多数的 Unix 实践中 [Johnson, 1975]。它被用来开发便捷的 C 编译程序 [Johnson, 1978] 以及其他许多编译程序。Bison 是由 Richard Stallman 及其他人一同开发的；Gnu Bison 是 Free Software Foundation 的 Gnu 软件分配的一部分且可在许多 Internet 站点上得到。Yacc 用法的一个示例是在 Kernighan 和 Pike [1984] 中开发的有用但简洁的计算器程序。在 Schreiner 和 Friedman [1985] 中可找到有关 Yacc 用法的完整研究。

LR 错误校正技术是 Graham、Haley 和 Joy [1979]、Penello 和 DeRemer [1978] 和 Burke 和 Fisher [1987] 中的研究内容。在 Fischer 和 LeBlanc [1991] 中描述了一个 LR 错误修正技术。5.7.2 节中描述的应急方式技术由 Fischer 和 LeBlanc 归结于 James [1972] 中。

China-pub.com

下载

第6章 语义分析

本章要点

- 属性和属性文法
- 数据类型和类型检查
- 属性计算算法
- TINY 语言的语义分析
- 符号表

本章要研究的编译程序阶段是计算编译过程所需的附加信息。这个阶段称作语义分析，因为它包括计算上下文无关文法和标准分析算法以外的信息，因此，它不被看成是语法^①。信息的计算也与被翻译过程的最终含义或语义密切相关。因为编译器完成的分析是静态（它在执行之前发生）定义的，这样，语义分析也可称作静态语义分析（static semantic analysis）。在一个典型的静态类型的语言（如C语言）中，语义分析包括构造符号表、记录声明中建立的名字的含义、在表达式和语句中进行类型推断和类型检查以及在语言的类型规则作用域内判断它们的正确性。

语义分析可以分为两类。第1类是程序的分析，要求根据编程语言的规则建立其正确性，并保证其正确执行。对于不同的语言来说，语言定义所要求的这一类分析的总量变化很大。在LISP和Smalltalk这类动态制导的语言中，可能完全没有静态语义分析；而在Ada这类语言中就有很强需求，程序必须提交执行。其他的语言介于这两种极端情况之间（例如Pascal语言，不像Ada和C对静态语义分析的要求那样严格，也不像LISP那样完全没有要求）。

语义分析的第2类是由编译程序执行的分析，用以提高翻译程序执行的效率。这一类分析通常包括对“最优化”或代码改进技术的讨论。第8章“代码生成”中将研究一些这样的方法，而本章则集中讨论语言定义对正确性要求的一般分析。读者应该注意到，这里研究的技术对两类情况都适用。这两类分析也不是相互排斥的，因为与没有正确性要求的语言相比，如静态类型检查这样的正确性要求能使编译程序产生更加有效的代码。另外，值得注意的是，这里讨论的正确性要求永远不能建立程序的完全正确性，正确性仅仅是部分的。但这样的要求仍然是有用的，可以给编程人员提供一些信息，提高程序的安全性和有效性。

静态语义分析包括执行分析的描述（description）和使用合适的算法对分析的实现（implementation）。在这里，它和词法及语法分析相类似。例如，在语法分析中，我们使用Backus-Naur范式(BNF)中的上下文无关文法描述语法结构，并用各种自顶向下和自底向上的分析算法实现语法结构。在语义分析中，情形不是那么清晰，其部分原因是没有用标准的方法（如BNF）来说明语言的静态语义；另一个原因是对于各种语言，静态语义分析的种类和总量的变化范围很大。编译程序编写者过去常用的且实现得很好的一种描述语义分析方法是：确定语言实体的属性(attribute)或特性，它们必须进行计算并写成属性等式(attribute equation)或语义规则(semantic rule)，并描述这些属性的计算如何与语言的文法规则相关。这样的一组属性和等式称作属性文法(attribute grammar)。属性文法对遵循语法制导语义(syntax-directed semantic)原

① 这一点在第3章的3.6.3节进行了较为详细的讨论。

理的语言最有用，它表明程序的语义内容与它的语法密切相关。所有的现代语言都有这个特性。然而，编译程序的编写者通常必须根据语言手册手工构造属性文法，因为语言设计者很少为之提供。更糟糕的是，由于坚持语言清晰的语法结构，属性文法的构造会有不必要的复杂性。语义计算表达式的一种更好标准是抽象语法，就像抽象语法树表示的那样。但是抽象语法树的说明通常也由语言设计者留给了编译程序编写者。

语义分析实现的算法也不像语法分析算法那样能清晰地表达。这部分原因也是因为在考虑语义分析说明时，出现了刚刚提及的同样的问题。还有另外一个问题，它是由编译过程中分析的时间选择引起的。如果语义分析可以推迟到所有的语法分析（以及抽象语法树的构造）完成之后进行，那么实现语义分析的任务就相当容易，其本质上由指定对语法树遍历的一个顺序组成，同时在遍历中每次遇到节点时进行计算。这就意味着编译程序必须是多遍的。另一方面，如果必须要求编译程序在一遍中完成所有的操作（包括代码生成），那么语义分析的实现就更加会变成寻找计算语义信息的正确顺序和方法的特别的过程（假定这样的顺序实际存在）。当然，现代的惯例越来越允许编译程序编写者使用多遍扫描简化语义分析和代码生成的过程。

尽管有点扰乱语义分析的状态，研究属性文法和规范发布仍是特别有用的，因为这能从写出更加清晰、简练、不易出错的语义分析代码中得到补偿，同时代码也更加易懂。

因此，本章从研究属性和属性文法开始。接下来是通过属性文法说明实现计算的技术，包括推断与树的遍历相连的计算顺序。随后的两节集中于语义分析的两个主要方面：符号表和类型检查。最后一节讲述前一章介绍的TINY编程语言的语义分析程序。

与第5章不同，本章没有包含对语义分析程序生成器 (semantic analyzer generator) 或构造语义分析程序的通用工具的描述。尽管已经构造了许多这样的工具，但没有一个能得到广泛的使用且能用于Lex或Yacc。在本章最后的“注意与参考”一节中，我们提及了几个这样的工具，并为感兴趣的读者提供了参考文献。

6.1 属性和属性文法

属性(attribute)是编程语言结构的任意特性。属性在其包含的信息和复杂性等方面变化很大，特别是当它们能确定时翻译/执行过程的时间。属性的典型例子有：

- 变量的数据类型。
- 表达式的值。
- 存储器中变量的位置。
- 程序的目标代码。
- 数的有效位数。

可以在复杂的处理（甚至编译程序的构造）之前确定属性。例如，一个数的有效位数可以根据语言的定义确定（或者至少给出一个最小值）。属性也可以在程序执行期间才确定，如（非常数）表达式的值，或者动态分配的数据结构的位置。属性的计算及将计算值与正在讨论的语言结构联系的过程称作属性的联编(binding)。联编属性发生时编译/执行过程的时间称作联编时间(binding time)。不同的属性变化，甚至不同语言的相同属性都可能完全有不同的联编时间。在执行之前联编的属性称作静态的(static)，而只在执行期间联编的属性是动态的(dynamic)。对于编译程序编写者而言，当然对那些在翻译时联编的动态属性感兴趣。

考虑先前给出的属性表的示例。我们讨论表中每个属性在编译时的联编时间和重要性。

- 在如C或Pascal这样的静态类型的语言中，变量或表达式的数据类型是一个重要的编译时属性。类型检查器(type checker)是一个语义分析程序，它计算定义数据类型的所有语言

实体的数据类型属性，并验证这些类型符合语言的类型规则。在如 C 或 Pascal 这样的语言中，类型检查是语义分析的一个重要部分。而在 LISP 这样的语言中，数据类型是动态的，LISP 编译程序必须生成代码来计算类型，并在程序执行期间完成类型检查。

- 表达式的值通常是动态的，编译程序要在执行时生成代码来计算这些值。然而事实上，一些表达式可能是常量（例如 $3+4*5$ ），语义分析程序可以选择在编译时求出它们的值（这个过程称作常量合并(constant folding)）。
- 变量的分配可以是静态的也可以是动态的，这依赖于语言和变量自身的特性。例如，在 FORTRAN77 中所有的变量都是静态分配的，而在 LISP 中所有的变量是动态分配的。C 和 Pascal 语言混合了静态和动态的两种变量分配。因为编译程序与变量分配联系的计算依赖于运行时环境，有时还依赖于具体的目标机器，所以这些计算会一直推迟到代码生成（第 7 章将更详细地探讨这一问题）。
- 程序的目标代码无疑是一个静态属性。编译程序的代码生成器专门用于这个属性的计算。
- 数的有效位数在编译期间是一个不被明确探讨的属性。编译程序编写者实现值的表示是不言而喻的，这通常被视为运行时环境的一部分，这将在第 7 章中讨论。然而，甚至扫描器也需要知道允许的有效位数并判断是否正确转换了常量。

正如从这些例子中看到的，属性计算变化极大。当它们在编译程序中显式出现时，可能在编译过程的任意时刻发生：即使语义分析阶段与属性计算的联系最紧密，扫描器和语法分析程序也都需要对它们有用的属性信息，在语法分析的同时也需要进行一些语义分析。在本章中，我们集中讨论在代码生成前及语法分析后的典型计算（语法分析期间的语义分析信息见 6.2.5 节）。直接应用于代码生成的属性分析在第 8 章中讨论。

6.1.1 属性文法

在语法制导语义(syntax-directed semantics)中，属性直接与语言的文法符号相联系（终结符号或非终结符号）^①。如果 X 是一个文法符号， a 是 X 的一个属性，那么我们把与 X 关联的 a 的值记作 $X.a$ 。这个记号让人回忆起 Pascal 语言的记录字段表示符或（等价于）C 语言中的结构成员操作符。实际上，实现属性计算的一种典型方法是使用记录字段（或结构成员）将属性值放到语法树的节点中去。下一节将更详细地讨论这一点。

若有一个属性的集合 a_1, \dots, a_k ，语法制导语义的原理应用于每个文法规则 $X_0 \rightarrow X_1 X_2 \dots X_n$ （这里 X_0 是一个非终结符号，其他的 X_i 都是任意符号），每个文法符号 X_i 的属性 $X_i.a_j$ 的值与规则中其他符号的属性值有关。如果同一个符号 X_i 在文法规则中出现不止一次，那么每次必须用合适的下标与在其他地方出现的符号区分开来。每个关系用属性等式(attribute equation)或语义规则(semantics rule)^② 表示，形式如下

$$X_i.a_j = f_{ij}(X_0.a_1, \dots, X_0.a_k, X_1.a_1, \dots, X_1.a_k, \dots, X_n.a_1, \dots, X_n.a_k)$$

这里的 f_{ij} 是一个数学函数。属性 a_1, \dots, a_k 的属性文法(attribute grammar)是对语言的所有文法规则的所有这类等式的集合。

在这个一般性情况中，属性文法显得相当复杂。在实际情况下，函数 f_{ij} 通常非常简单。属性也很少依赖于大量的其他属性，因此可以将相互依赖的属性分割为较小的独立属性集，并对

① 语法制导语义可以简单地称作语义制导语法(semantics-directed syntax)，因为在大多数语言中语法是用已经在头脑中建立的(最终)语义设计的。

② 后面我们将看到语义规则比属性等式更加通用一些。这里读者可以先把它们看成是等效的。

每个属性集单独写出一个属性文法。

一般地，将属性文法写成表格形式，每个文法规则用属性等式的集合或者相应规则的语义规则列出，如下所示^①：

文 法 规 则	语 义 规 则
规则1	相关的属性等式
.	.
.	.
.	.
规则 n	相关的属性等式

接下来继续看几个例子。

例6.1 考虑下面简单的无符号数文法：

$$\begin{aligned} \text{number} & \quad \text{number digit} \mid \text{digit} \\ \text{digit} & \quad 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{aligned}$$

一个数最重要的属性是它的值，我们将其命名为 *val*。每个数字都有一个值，可以用它表示的实际数直接计算。因此，例如，文法规则 *digit 0* 表明在这个情况下 *digit* 的值为0。这可以用属性等式 *digit.val = 0* 表示，我们将这个等式和规则 *digit 0* 联系在一起。此外，每个数都有一个基于它所包含的数字的值。如果一个数使用了下面的规则推导

$$\text{number} \quad \text{digit}$$

那么这个数就只包含了一个数字，其值就是这个数字的值。用属性等式表示为

$$\text{number.val} = \text{digit.val}$$

如果一个数包含的数字多于1个，可以使用下列文法规则推导

$$\text{number} \quad \text{number digit}$$

我们必须表示出这个文法规则左边符号的值和右边符号的值之间的关系。请读者注意，在这个文法规则中对 *number* 的两次出现必须进行区分，因为右边的 *number* 和左边的 *number* 的值不相同。我们使用下标进行区分，将这个文法写成如下形式：

$$\text{number}_1 \quad \text{number}_2 \text{ digit}$$

现在考虑一个数，如 34。这个数的(最左)推导如下：*number number digit digit 3 digit 34*。考虑在这个推导的第一步文法规则 *number₁ number₂ digit* 的使用。非终结符号 *number₂* 对应于数字 3，*digit* 对应于数字 4。每个符号的值分别是 3 和 4。为了获得 *number₁* 的值(它为 34)，必须用 *number₂* 的值乘以 10，再加上 *digit* 的值： $34 = 3 * 10 + 4$ 。换句话说，是把 3 左移一个十进制位再加上 4。相应的属性等式是

$$\text{number}_1.\text{val} = \text{number}_2.\text{val} * 10 + \text{digit.val}$$

完整的 *val* 属性文法在表 6-1 中给出。

使用字符串的语法树可以形象化地表示特殊字符串的属性等式的意义。例如，图 6-1 给出了数 345 的语法树。在这个图中相应合适的属性等式的计算显示在每个内部节点的下面。在语

^① 这些表中通常用表头“语义规则”代替“属性等式”，以便后面对语义规则进行更通用的解释。

法树中计算时观察属性等式对于计算属性值的算法是重要的，就像在下一节将看到的那样[⊖]。

在表6-1和图6-1中，通过使用不同的字体，我们强调了数字和值的语法表示或者数字语义内容之间的不同。例如在文法规则 *digit* 0 中，数字0是一个记号或特性，而 *digit.val* = 0 的含义是数字的数值为0。

表6-1 例6.1的属性文法

文法规则	语义规则
<i>Number</i> ₁ <i>number</i> ₂ <i>digit</i>	<i>number</i> ₁ .val = <i>number</i> ₂ .val * 10 + <i>digit</i> .val
<i>Number</i> <i>digit</i>	<i>number</i> .val = <i>digit</i> .val
<i>digit</i> 0	<i>digit</i> .val = 0
<i>digit</i> 1	<i>digit</i> .val = 1
<i>digit</i> 2	<i>digit</i> .val = 2
<i>digit</i> 3	<i>digit</i> .val = 3
<i>digit</i> 4	<i>digit</i> .val = 4
<i>digit</i> 5	<i>digit</i> .val = 5
<i>digit</i> 6	<i>digit</i> .val = 6
<i>digit</i> 7	<i>digit</i> .val = 7
<i>digit</i> 8	<i>digit</i> .val = 8
<i>digit</i> 9	<i>digit</i> .val = 9

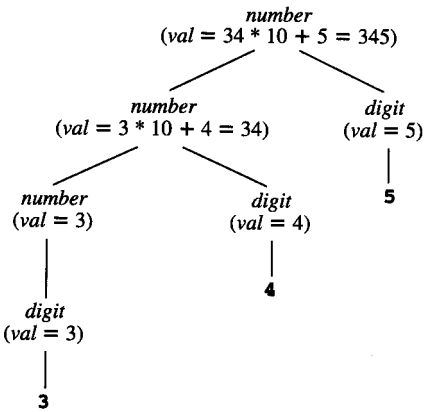


图6-1 显示例6.1中属性计算的语法树

例6.2 考虑下列简单的整数算术表达式文法：

exp *exp* + *term* | *exp* - *term* | *term*
term *term* * *factor* | *factor*
factor (*exp*) | **number**

这个文法对第5章广泛讨论的简单表达式文法稍微作了一些改动。*exp*(或*term*或*factor*)的基本属性是它的数字值，写作*val*。*val* 属性的属性等式在表6-2中给出。

⊖ 事实上，扫描器通常认为数是标记，它们的数值也很容易计算。在此期间，扫描器很可能隐含地使用这里定义的属性等式。

表6-2 例6.2的属性文法

文法规则	语义规则
$exp_1 \rightarrow exp_2 + term$	$exp_1.val = exp_2.val + term.val$
$exp_1 \rightarrow exp_2 - term$	$exp_1.val = exp_2.val - term.val$
$exp \rightarrow term$	$exp.val = term.val$
$term_1 \rightarrow term_2 * factor$	$term_1.val = term_2.val * factor.val$
$term \rightarrow factor$	$term.val = factor.val$
$factor \rightarrow (exp)$	$factor.val = exp.val$
$factor \rightarrow \text{number}$	$factor.val = \text{number.val}$

这些等式表示了表达式的语法和它所进行的算术运算的语义之间的关系。注意，在文法规则

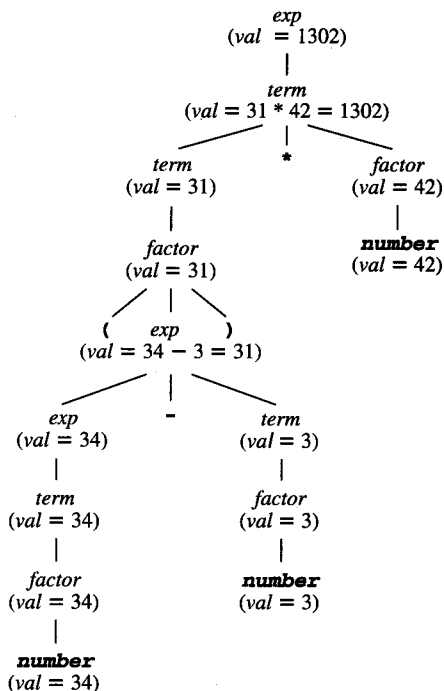
$$exp_1 \rightarrow exp_2 + term$$

中语义符号 + (记号) 和等式

$$exp_1.val = exp_2.val + term.val$$

中执行的算术加运算符 + 的不同。还要注意 **number.val** 不会在等式的左边。就像在下一节将看到的，这意味着必须在任意一个使用这个属性文法（例如扫描器）的语义分析之前计算 **number.val**。换句话说，如果想在属性文法中明确这个值，就必须在属性文法中加进文法规则和属性等式（例如，例6.1中的等式）。

如例6.1一样，也可以通过在语法树的节点上附加等式来表示属性文法包含的计算。例如，给定表达式 $(34-3)*42$ ，可以用在其语法树上值的语义来表达，如图6-2所示。

图6-2 $(34-3)*42$ 的语法树，显示例6.2中属性文法的val属性计算

例6.3 考虑下列类似C语法中变量声明的简单文法：

```
decl    type var-list
type    int | float
var-list id, var-list | id
```

我们要为在声明中标识符给出的变量定义一个数据类型属性，并写出一个等式来表示数据类型属性是如何与声明的类型相关的。通过构造 *dtype*属性的一个属性文法可以实现这一点(使用名字*dtype*和非终结符*type*的属性进行区分)。*dtype*的属性文法在表6-3中给出。在图中关于属性等式我们做了以下标记。

首先，从{integer, real}集合中得出*dtype*的值，相应的记号为int和float。非终结符*type*有一个它表示的记号给定的 *dtype*。通过*decl*文法规则的等式，这个 *dtype*对应于全体*var-list*的*dtype*。通过*var-list*的等式，表中的每个 *id*都有相同的*dtype*。注意，没有等式包含非终结符*decl* 的*dtype*。实际上*decl*并不需要*dtype*，一个属性的值没有必要为所有的文法符号指定。

同前面一样，可以在一个语法树上显示属性等式。图 6-3给出了一个例子。

表6-3 例6.3的属性文法

文法规则	语义规则
<i>decl</i> <i>type</i> <i>var-list</i>	<i>var-list.dtype</i> = <i>type.dtype</i>
<i>type</i> int	<i>type.dtype</i> = integer
<i>type</i> float	<i>type.dtype</i> = real
<i>var-list</i> ₁ id , <i>var-list</i> ₂	id.dtype = <i>var-list</i> _{1.dtype}
	<i>var-list</i> _{2.dtype} = <i>var-list</i> _{1.dtype}
<i>var-list</i> id	id.dtype = <i>var-list.dtype</i>

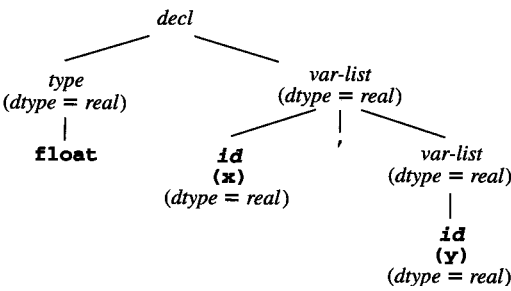


图6-3 字符串float x,y的语法树，显示表6-3中属性文法指定的dtype属性

到目前为止，所有的例子都只有一个属性，但属性文法可能会包含几个独立的属性。下一个例子是一个有几个相互联系属性的简单情形。

例6.4 考虑对例6.1中数文法进行修改，数可以是八进制或十进制的。假设这通过一个字符的后缀o(八进制)或d(十进制)来表示。这样就有下面的文法：

```
based-num    num basechar
basechar     o | d
num          num digit | digit
digit        0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

在这种情况下，*num*和*digit*均需要一个新的属性*base*用来计算属性*val*。*base*和*val*的属性文法在表6-4中给出。

表6-4 例6.4的属性文法

文法规则	语义规则
<i>based-num</i>	$Based\text{-}num.val = num.val$
<i>num basechar</i>	$num.base = basechar.base$
<i>basechar</i> o	$basechar.base = 8$
<i>basechar</i> d	$basechar.base = 10$
$num_1 \quad num_2 \quad digit$	$num_1.val =$ if $digit.val = error$ or $num_2.val = error$ then $error$ else $num_2.val * num_1.base + digit.val$ $num_2.base = num_1.base$ $digit.base = num_1.base$ $num.val = digit.val$ $digit.base = num.base$
<i>num digit</i>	$num.val = digit.val$ $digit.base = num.base$
<i>digit</i> 0	$digit.val = 0$
<i>digit</i> 1	$digit.val = 1$
...	...
<i>digit</i> 7	$digit.val = 7$
<i>digit</i> 8	$digit.val =$ if $digit.base = 8$ then $error$ else 8
<i>digit</i> 9	$digit.val =$ if $digit.base = 8$ then $error$ else 9

在这个属性文法中必须注意两个新的特性。首先，这个BNF文法在后缀为**o**时自己不能排除错误的(非八进制)数字8和9的组合。例如，按照上面的BNF文法，字符串**189o**在语法上是

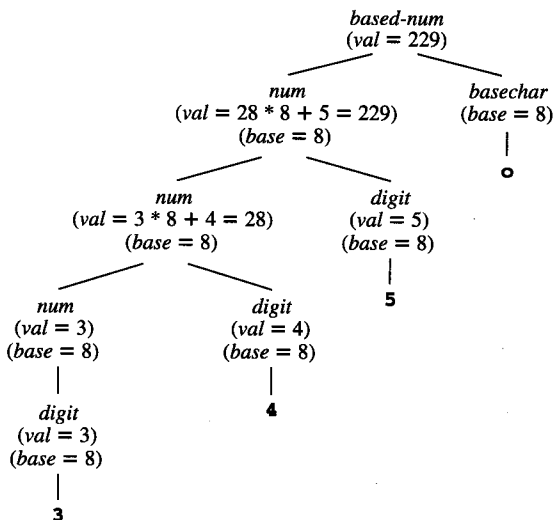


图6-4 例6.4中属性计算的语法树

正确的，但不能赋予任何值。因此，对于这些情况需要一个新的 *error* 值。另外，这个属性文法必须能表示这样的事实，一个带有后缀 *o* 的数中包括数字 8 或 9 时将导致值为 *error*。最简单的方法是在合适的属性等式函数中使用 *if-then-else* 表达式。例如，等式

$$\begin{aligned} \text{num}_1.\text{val} = & \\ & \text{if } \text{digit}.\text{val} = \text{error} \text{ or } \text{num}_2.\text{val} = \text{error} \\ & \text{then error} \\ & \text{else } \text{num}_2.\text{val} * \text{num}_1.\text{base} + \text{digit}.\text{val} \end{aligned}$$

对应于文法规则 $\text{num}_1 \rightarrow \text{num}_2 \text{ digit}$ ，表示如果 $\text{num}_2.\text{val}$ 或 $\text{digit}.\text{val}$ 为 *error*，那么 $\text{num}_1.\text{val}$ 也必须为 *error*，除此之外只有一种情况： $\text{num}_1.\text{val}$ 的值由公式 $\text{num}_2.\text{val} * \text{num}_1.\text{base} + \text{digit}.\text{val}$ 给出。

总结这个例子，再在一个语法树上表示属性计算。图 6-4 给出了数 3450 的语法树，同时根据表 6-4 的属性文法计算出属性值。

6.1.2 属性文法的简化和扩充

if-then-else 表达式的使用扩充了表达式的种类，它们可以通过有用的途径出现在属性等式中，在属性等式中，允许出现的表达式的集合称作属性文法的元语言 (metalanguage)。通常我们希望元语言的内涵尽可能清晰，不致于引起其自身语义的混淆。我们还希望元语言接近于一种实际使用的编程语言，因为就像我们即将看到的一样，在语义分析程序中需要把属性等式转换成执行代码。在本书中，我们使用的元语言局限于算术式、逻辑式以及一些其他种类的表达式，再加上 *if-then-else* 表达式，偶尔还有 *case* 或 *switch* 表达式。

定义属性等式的另一个有用的特征是在元语言中加入了函数的使用，函数的定义可以在别处给出。例如，在数的文法中，我们对 *digit* 的每个选择都写出了属性等式。可以采用一个更简洁的惯例来代替，为 *digit* 写一个文法规则 $\text{digit} \rightarrow D$ (这里 *D* 是数字中的一个)，相应的属性等式是

$$\text{digit}.\text{val} = \text{numval}(D)$$

这里 *numval* 是函数，必须在别处说明其作为属性文法的补充的定义。例如，我们可以用 C 代码给出 *numval* 的定义：

```
int numval ( char D )
{ return (int)D - (int)'0'; }
```

在说明属性文法时更简化的有用方法是使用原始文法二义性的但简单的形式。事实上，因为假定已经构造了分析程序，所有二义性在那个阶段都已经处理过了，属性文法可以自由地基于二义性概念，而无须在结果属性中说明任何二义性，例如，例 6.2 的算术表达式文法有以下简单的但二义性的形式：

$$\text{exp} \rightarrow \text{exp} + \text{exp} \mid \text{exp} - \text{exp} \mid \text{exp} * \text{exp} \mid (\text{exp}) \mid \text{number}$$

使用这个文法，属性 *val* 可以通过表 6-5 中的表定义 (与表 6-2 相比较)。

表 6-5 使用二义性文法定义表达式的 *val* 属性

文法规则	语义规则
$\text{exp}_1 \rightarrow \text{exp}_2 + \text{exp}_3$	$\text{exp}_1.\text{val} = \text{exp}_2.\text{val} + \text{exp}_3.\text{val}$
$\text{exp}_1 \rightarrow \text{exp}_2 - \text{exp}_3$	$\text{exp}_1.\text{val} = \text{exp}_2.\text{val} - \text{exp}_3.\text{val}$
$\text{exp}_1 \rightarrow \text{exp}_2 * \text{exp}_3$	$\text{exp}_1.\text{val} = \text{exp}_2.\text{val} * \text{exp}_3.\text{val}$
$\text{exp}_1 \rightarrow (\text{exp}_2)$	$\text{exp}_1.\text{val} = \text{exp}_2.\text{val}$
$\text{exp} \rightarrow \text{number}$	$\text{exp}.\text{val} = \text{number}.\text{val}$

在显示属性值时，也可以通过使用抽象语法树代替分析树进行简化。抽象语法树通常必须用足够的结构来表达属性文法定义的语义。例如表达式 $(34-3)*42$ ，其分析树和 *val* 属性在图 6-2 中已给出，通过图 6-5 的抽象语法树也能完全表达它的语义。

如下一个例子所示，语法树自身也可以用属性文法指定，这并不奇怪。

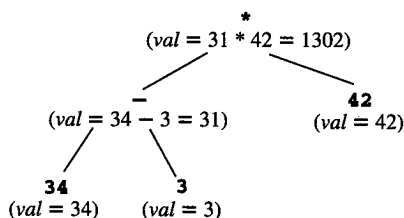


图6-5 $(34-3)*42$ 的抽象语法树，显示表 6-2 或表 6-5 中属性文法的 *val* 属性计算

例6.5 给定例 6.2 的简单整数表达式文法，通过表 6-6 给出的属性文法可以定义表达式的抽象语法树。在这个属性文法中，我们使用了两个辅助函数 *mkOpNode* 和 *mkNumNode*。*mkOpNode* 函数有 3 个参数(一个操作符记号和两个语法树)并构成一个新的树节点，其操作符记号是第 1 个参数，子节点是第 2 个和第 3 个参数。*mkNumNode* 函数有一个参数(数字值)并构成一个叶子节点，它表示具有这个值的一个数。在表 6-6 中我们把数字值写作 *number.lexval*，表示它是由扫描器构成的。事实上，根据不同的实现，这可以是实际的数字值，或用它的字符串表示(比较表 6-6 中的等式和附录 B 中 TINY 语法树递归下降构造)。

表6-6 简单整型算术表达式的抽象语法树的属性文法

文法规则	语义规则
$exp_1 \rightarrow exp_2 + term$	$exp_1.tree =$ $mkOpNode(+, exp_2.tree, term.tree)$
$exp_1 \rightarrow exp_2 - term$	$exp_1.tree =$ $mkOpNode(-, exp_2.tree, term.tree)$
$exp \rightarrow term$	$exp.tree = term.tree$
$term_1 \rightarrow term_2 * factor$	$term_1.tree =$ $mkOpNode(*, term_2.tree, factor.tree)$
$term \rightarrow factor$	$term.tree = factor.tree$
$factor \rightarrow (exp)$	$factor.tree = exp.tree$
$factor \rightarrow number$	$factor.tree =$ $mkNumNode(number.lexval)$

如何确保一个特定的属性文法是一致的和完整的，是使用属性文法的属性说明的一个主要问题，也就是说，它能唯一定义给定的属性。简单的答案是到目前为止还不能做到。这个问题与确定一个文法是否是二义的类似。实际上，这是用来确定一个文法充分性的分析算法，类似的情形发生在属性文法的情况中。因此，下一节将研究属性计算的算法规则方法，确定一个属性文法是否足够能定义属性值。

6.2 属性计算算法

这一节将以属性文法为基础，研究编译程序中如何计算和使用由属性文法的等式定义的属

性。基本上，这等于将属性等式转化为计算规则。因此，属性等式

$$X_i.a_j = f_{ij}(X_0.a_1, \dots, X_0.a_k, X_1.a_1, \dots, X_1.a_k, \dots, X_n.a_1, \dots, X_n.a_k)$$

被看作把右边的函数表达式的值赋给属性 $X_i.a_j$ 。为达到这一点，在右边出现的所有属性值必须已经存在。在属性文法的说明中，这个要求可能被忽略，可以将等式写成任意顺序而不会影响正确性。实现对应于属性文法的算法规则的问题，主要在于为赋值和属性分配寻找一个顺序，并确保当每次进行计算时使用的所有属性值都是有效的。属性等式本身指示了属性计算时的顺序约束，首先是使用指示图表示这些约束来明确顺序的约束。这些指示图叫做相关图。

6.2.1 相关图和赋值顺序

给定一个属性文法，每个文法规则选择一个相关依赖图 (associated dependency graph)。文法规则中的每个符号在这个图中都有用每个属性 $X_i.a_j$ 标记的节点，对每个属性等式

$$X_i.a_j = f_{ij}(\dots, X_m.a_k, \dots)$$

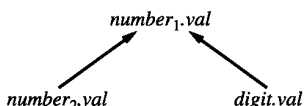
相关于文法规则从在右边的每个节点 $X_m.a_k$ 到节点 $X_i.a_j$ 有一条边(表示 $X_i.a_j$ 对 $X_m.a_k$ 的依赖)。依据上下文无关文法，在语言产生时给定一个合法的字符串，这个字符串的依赖图 (dependency graph) 就是字符串语法树中选择表示每个(非叶子)节点文法规则依赖图的联合。

在绘制每个文法规则或字符串的依赖图时，与每个符号 X 相关的节点均画在一组中，这样依赖就可以看作是语法树的构造。

例6.6 考虑例6.1的文法，属性文法在表 6-1 中给出。这里只有一个属性 val ，因此每个符号在每个依赖图中只有一个节点对应于其 val 属性。选择 $number_1 \quad number_2 \text{ digit}$ 的文法规则有单个的相关属性等式

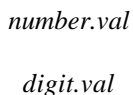
$$number_1.val = number_2.val * 10 + digit.val$$

这个文法规则选择的依赖图是



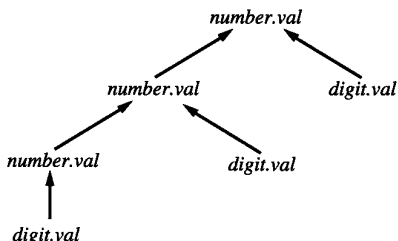
(在以后的依赖图中，因为图形化表示可以清楚地区分不同节点的不同出现，所以对重复的符号将省略下标)。

类似地，属性等式 $number.val = digit.val$ 中文法规则 $number \rightarrow digit$ 的相关图是



对于剩下的形如 $digit \rightarrow D$ 的文法规则，因为 $digit.val$ 可以从规则的右边直接计算，其相关图是无足轻重的(它们没有边)。

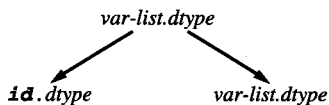
最后，对应于语法树(见图6-1)，字符串 345 的相关图如下：



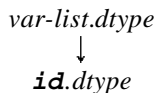
例6.7 考虑例6.3的文法，属性 *dtype* 的属性文法在表 6-3 中给出。在这个例子中，文法规则 $var-list_1 \rightarrow id, var-list_2$ 有两个相关的属性等式

$$\begin{aligned} id.dtype &= var-list_1.dtype \\ var-list_2.dtype &= var-list_1.dtype \end{aligned}$$

依赖图是

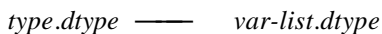


类似地，文法规则 $var-list \rightarrow id$ 的相关图是

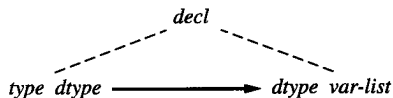


$type \rightarrow int$ 和 $type \rightarrow float$ 两个规则的相关图无关紧要。

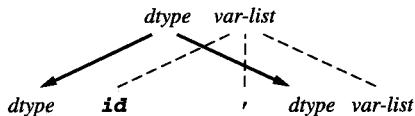
最后，与等式 $var-list.dtype = type.dtype$ 相关的 $decl \rightarrow type \mid var-list$ 规则的相关图是



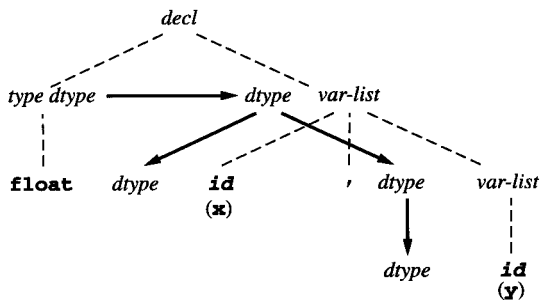
在这种情况下，因为 *decl* 不直接包含在相关图中，所以这个相关图相关的文法规则并不完全清晰。正因为这一点（稍后将讨论其他一些原因），我们通常重叠在相应的文法规则的语法树片段上绘制相关图。这样，上面的相关图就可以画作



这就使和相关有关的文法规则更加清晰。还要注意，在画语法树节点时我们禁止使用属性的圆点符号，而是通过写出与其相连的下一个节点来表示属性。这样，在这个例子中第一个相关图也可以画作



最后，字符串 `float x,y` 的相关图是

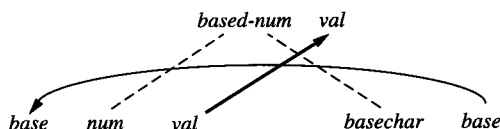


例6.8 考虑例6.4基于数的文法，属性 $base$ 和 val 的属性文法在表6-4中给出。画出下面4条文法规则

$based_num \quad num \ basechar$
 $num \quad num \ digit$
 $num \quad digit$
 $digit \quad 9$

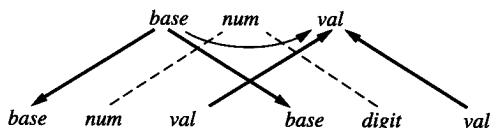
和字符串3450的相关图，其语法树在图6-4中给出。

首先从文法规则 $based_num \quad num \ basechar$ 的相关图开始：



这个图表示了 $based_num.val = num.val$ 和 $num.base = basechar.base$ 两个相关的等式的相关。

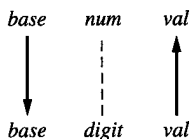
接下来再画出与文法规则 $num \quad num \ digit$ 相对应的相关图：



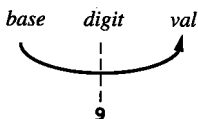
这个图表示3个属性等式的相关

$num_1.val =$
if $digit.val = error$ **or** $num_2.val = error$
then $error$
else $num_2.val * num_1.base + digit.val$
 $num_2.base = num_1.base$
 $digit.base = num_1.base$

文法规则 $num \quad digit$ 的相关图也与此类似：



最后，画出文法规则 $digit \quad 9$ 的相关图：



这个图表示由等式 $digit.val = \text{if } digit.base = 8 \text{ then } error \text{ else } 9$ 创建的相关，也就是 $digit.val$ 与于 $digit.base$ (这是if表达式测试的一部分)相关。现在剩下画出字符串3450的相关图，如图6-6所示。

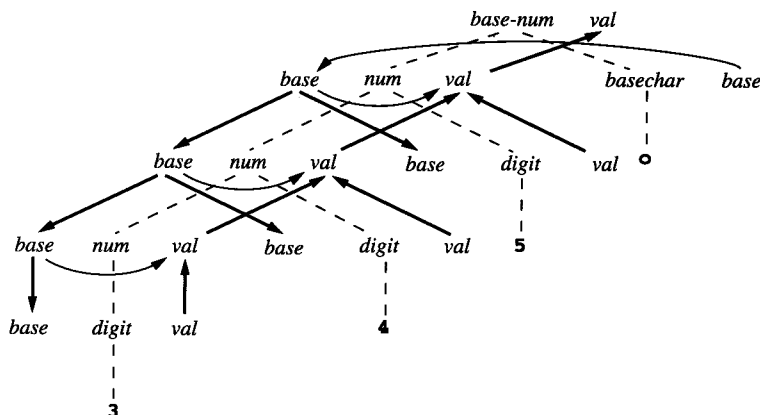


图6-6 字符串3450的相关图(例6.8)

假设现在要确定一个算法,使用属性等式作为计算规则来计算一个属性文法的属性。给定要转换的一个特定的记号字符串,字符串语法树的相关图根据计算字符串属性的算法给出了一系列顺序约束。实际上,任一个算法在试图计算任何后继节点的属性之前,必须计算相关图中每个节点的属性。遵循这个限制的相关图遍历顺序称作拓扑排序(topological sort),而且众所周知,存在拓扑排序的充分必要条件是相关图必须是非循环(acyclic)的。这样的图形称作确定非循环图(directed acyclic graphs, DAGs)。

例6.9 图6-6的相关图是一个DAG。在图6-7中,给节点编上号(为便于观察删除了下面的语法树)。根据节点的编号给出了一个拓扑排序的顺序。另一个拓扑排序的顺序是

12 6 9 1 2 11 3 8 4 5 7 10 13 14

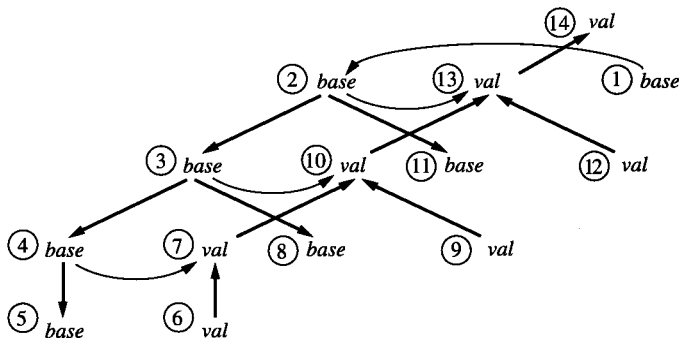


图6-7 字符串3450的相关图(例6.9)

在图的根节点上如何找到属性值(图的根节点是没有前驱的节点)是在使用相关图的拓扑排序计算属性值时产生的一个问题。在图6-7中,节点1、6、9、12都是图的根节点[⊖]。这些节点的属性值不依赖于任何其他属性,因此必须使用直接可用的信息计算。这个信息通常在相应的语法树节点的子孙记号表中。例如,在图6-7中,节点6的val依赖于记号3,它是对应于val的digit节点的子节点(见图6-6)。因此,节点6的属性值是3。所有这些根节点的值都需要在计算其他任何属性值之前计算。这些计算通常由扫描器或语法分析程序来完成。

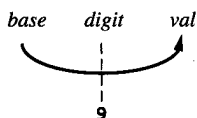
⊖ 相关图的根节点不要和语法树的根节点混淆。

在编译以及随后的为了确定属性赋值的顺序而进行的相关图的拓扑排序过程中, 基于一个算法在相关图的构造上进行属性分析是可能的。因为在编译时相关图的构造是基于特定的语法树的, 这个方法有时称作分析树方法 (parse tree method)。在任何非循环 (noncircular) 的属性文法中能够对属性赋值, 也就是说, 对每一个可能的相关图属性文法是非循环的。

这种方法有几个问题。首先, 在编译时构造相关图增加了额外的复杂性。其次, 这种方法在编译时确定相关图是否非循环时, 它通常不恰当地等待发现一个环, 直到到达编译时间, 因为在原始的属性文法中, 环几乎都是表示错误。换句话说, 属性文法必须预先进行无环测试。有一个算法进行这个工作 (参见“注意与参考”一节), 但这是一个时间指数级增长的算法。当然, 这个算法只需要在编译器构造时运行一次, 因此这不能成为反对这个算法的压倒性的证据 (至少对编译器的构造而言)。这种方法的复杂性就是更强的证据。

上述的属性赋值的另一种方法 (几乎每一个编译器都采用) 即编译器编写者在编译器构造时分析属性文法, 并固定一个属性赋值顺序。虽然这种方法仍然使用语法树作为属性赋值的指导, 但因为它依赖于对属性等式或语义规则的分析, 所以它仍被称作基于规则的方法 (rule-based method)。在编译器构造时固定属性赋值顺序的这一类属性文法没有所有的非循环属性文法通用, 但在实际中, 所有合理的属性文法都有这个特性。它们有时称作强非循环 (strongly noncircular) 属性文法。在下面的例子后面, 我们将对这类属性文法讨论基于规则的算法。

例6.10 再次考虑例6.8的相关图和例6.9中讨论的相关图的拓扑排序 (见图6-7)。虽然图6-7中的节点6、9和12是DAG的根节点, 并且因此都能够出现在拓扑排序的开始, 但在基于规则的方法中这是不可能的。其原因是如果相应的记号是8或9, 任何 *val* 可能依赖于与它相关的 *digit* 节点的 *base*。例如, 对 *digit* 9 相关图是



因此, 在图6-7中, 节点6可能依赖于节点5, 节点9可能依赖于节点8, 而节点12可能依赖于节点11。在基于规则的方法中, 这些节点将在任何可能潜在依赖的节点之后被强制赋值。因此, 一种赋值顺序, 首先赋值节点12 (见例6.9), 这对图6-7中特定的树是正确的, 但对基于规则的算法这是错误的顺序, 因为它将破坏其他语法树的顺序。

6.2.2 合成和继承属性

基于规则的属性赋值依赖于分析树或语法树明确或不明确的遍历。不同种类的遍历处理的属性相关, 在项目 and 能力的种类上都不同。为了研究这些不同之处, 首先必须根据它们具有的相关的种类对属性分类。处理的最简单的依赖是综合属性, 定义如下。

定义 一个属性是合成的 (synthesized), 如果在语法树中它所有的相关都从子节点指向父节点。等价地, 一个属性 a 是合成的, 如果给定一个文法规则 $A \rightarrow X_1 X_2 \dots X_n$, 左边仅有一个 a 的相关属性等式有以下形式

$$A.a = f(X_1.a_1, \dots, X_1.a_k, \dots, X_n.a_1, \dots, X_n.a_k)$$

一个属性文法中所有的属性都是合成的, 就称作S属性文法 (S-attributed grammar)。

我们已经见过了一些合成属性和S属性文法的例子。在例6.1中数的 *val* 属性是合成的 (参见

例6.6中的相关图), 例6.2中简单整数算术表达式的 *val* 属性也是一样。

给定由分析程序构造的分析树或语法树, *S* 属性文法的属性值可以通过对树进行简单的自底向上或后序遍历来计算。这可以用以下递归的属性赋值程序来表示:

```

procedure PostEval (T : treenode);
begin
    for each child C of T do
        PostEval (C);
    compute all synthesized attributes of T;
end;

```

例6.11 用合成属性 *val* 考虑例6.2中简单算术表达式的属性文法。给定下列语法树(就像图6-5)的结构

```

typedef enum { Plus, Minus, Times } OpKind;
typedef enum { OpKind, ConstKind } ExpKind;
typedef struct streenode
    { ExpKind kind;
      OpKind op;
      struct streenode *lchild, *rchild;
      int val;
    } STreeNode;
typedef STreeNode *SyntaxTree;

```

PostEval程序可以转换成程序清单6-1中的C程序代码, 进行从左向右的遍历。

程序清单6-1 例6.11中后序属性赋值的C程序代码

```

void postEval(SyntaxTree t)
{ int temp;
  if (t->kind == OpKind)
  { postEval(t->lchild);
    postEval(t->rchild);
    switch (t->op)
    { case Plus:
        t->val = t->lchild->val + t->rchild->val;
        break;
      case Minus:
        t->val = t->lchild->val - t->rchild->val;
        break;
      case Times:
        t->val = t->lchild->val * t->rchild->val;
        break;
    } /* end switch */
  } /* end if */
} /* end postEval */

```

当然, 不是所有的属性都是合成的。

定义 一个属性如果不是合成的, 则称作继承(inherited)属性。

我们已经见过继承属性的例子，包括例 6.3 中的 *dtype* 属性和例 6.4 中的 *base* 属性。无论在分析树中(使用这个名字的理由)从祖先到子孙的继承属性还是从同属的继承属性都有依赖。图 6-8 a 和 b 说明了继承属性依赖的两种基本种类。这两种类型的依赖在例 6.7 中的 *dtype* 属性中都出现过。这两种分类都为继承属性的原因在于计算继承属性的算法，同属继承通常是通过把属性值经过祖先在同属中传递实现的。实际上，如果语法树的边只从祖先指向子孙(这样子孙不能直接访问祖先或同属)这也是必要的。另一方面，在语法树中如果一些结构经过同属指针实现，那么同属继承可以直接沿着同属链进行，如图 6-8c 的描述。

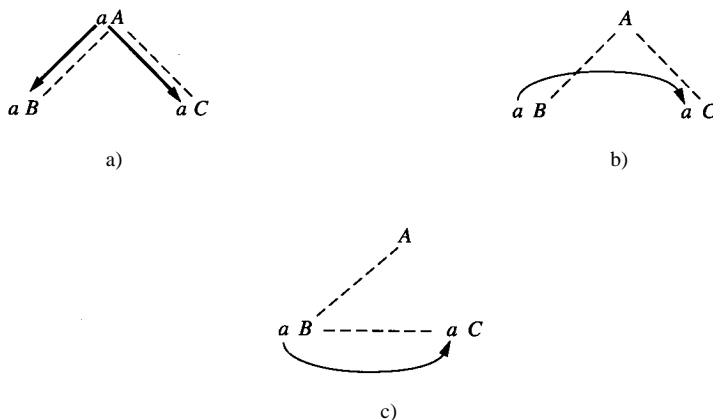


图6-8 不同种类的继承依赖

a) 从祖先到子孙的继承 b) 同属之间的继承 c) 通过同属指针的同属继承

现在回到继承属性赋值的算法方法。继承属性的计算可以通过对分析树或语法树的前序遍历或前序/中序遍历的组合来进行。这可以用下面的伪代码来示意表示：

```

procedure PreEval (T: treenode);
begin
  for each child C of T do
    compute all inherited attributes of C;
    PreEval (C);
end;

```

与合成属性不同，子孙继承属性计算的顺序是重要的，因为在子孙的属性中继承属性可能有依赖关系。因此在前面的伪代码中 *T* 的每个子孙 *C* 的访问顺序必须满足这些依赖的任何要求。在下面两个例子中，我们将使用前面例子中的 *dtype* 和 *base* 继承属性来说明这一点。

例6.12 考虑例6.3的文法，它有继承属性 *dtype*，其相关图例6.7中给出(见表6-3的属性文法)。首先假设从文法明确构造了分析树，为便于参考重述如下：

```

decl    type var-list
type    int | float
var-list id, var-list | id

```

所有需要的节点 *dtype* 属性的递归程序的伪代码如下：

```

procedure EvalType (T: treenode);

```

begin

case *nodekind of T of*

decl :

EvalType (type child of T);

Assign dtype of type child of T to var-list child of T;

EvalType (var-list child of T);

type :

if *child of T = int* **then** *T.dtype := integer*

else *T.dtype := real;*

var-list :

assign T.dtype to first child of T;

if *third child of T is not nil* **then**

assign T.dtype to third child;

EvalType (third child of T);

end case;

end *EvalType;*

注意，前序和中序操作是如何根据被处理的不同类型的节点混合的。例如，节点 *decl* 在子节点上递归调用 *EvalType* 之前，要求首先计算其第1个子孙的 *dtype*，然后分配给其第2个子孙；这是一个中序处理。另一方面，*var-list* 节点在进行任何递归调用之前分配 *dtype* 给其子孙；这是一个前序处理。

在图6-9中，我们与 *dtype* 属性的相关图一起显示了字符串 *float x, y* 的分析树，并且按照上面的伪代码计算 *dtype* 的顺序给节点编了号。

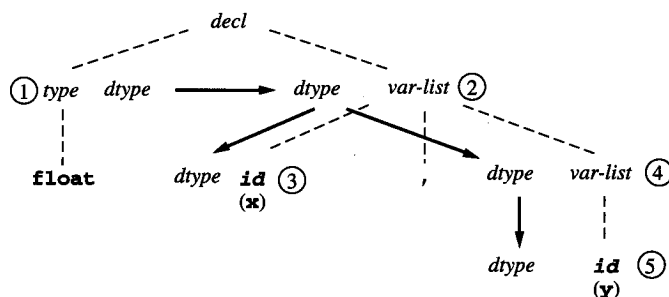
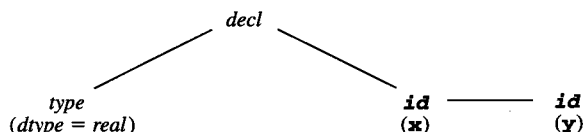


图6-9 显示例6.12遍历顺序的分析树

为了给出这个例子一个完全具体的形式，我们把前面的伪代码转换成实际的 C 语言代码。并且，为了替代使用明确的分析树，假设已经构造了一个语法树，*var-list* 用 *id* 节点的同属列表表示。这样，像 *float x, y* 这样的声明字符串语法树如下(与图6-9比较)



decl 节点的子孙赋值顺序为从左至右 (首先是节点 *type*，然后是节点 *x*，最后是节点 *y*)。注意，

在这个树中已经包括了 *dtype* 节点和 *type* 节点，假定在语法分析时已经预先计算了。

语法树的结构用下面的C语言声明给出：

```
typedef enum { decl, type, id } nodekind;
typedef enum { integer, real } typekind;
typedef struct treeNode
{
    nodekind kind;
    struct treeNode
    *lchild, *rchild, *sibling;
    typekind dtype;
    /* for type and id nodes */
    char * name;
    /* for id nodes only */
} * SyntaxTree;
```

相应地 *EvalType* 程序的C代码如下：

```
void evalType(SyntaxTree t)
{
    switch (t->kind)
    {
        case decl:
            t->rchild->dtype = t->lchild->dtype;
            evalType(t->rchild);
            break;
        case id:
            if (t->sibling != NULL);
            {
                t->sibling->dtype = t->dtype;
                evalType(t->sibling);
            }
            break;
    } /* end switch */
} /* end evalType */
```

这段代码可以用下列非递归程序简化，其操作全部在根节点 (*decl*) 层次上进行：

```
void evalType(SyntaxTree t)
{
    if (t->kind == decl)
    {
        SyntaxTree p = t->rchild;
        p->dtype = t->lchild->dtype;
        while (p->sibling != NULL)
        {
            p->sibling->dtype = p->dtype;
            p = p->sibling;
        }
    } /* end if */
} /* end evalType */
```

例6.13 考虑例6.4的文法，它具有继承属性 *base*（相关图在例6.8中给出）。这里将那个例子的文法重述如下：

```
based-num    num basechar
basechar    o | d
num         num digit | digit
```

9|8|7|6|5|4|3|2|1|0

这个文法有两个新特性。首先，它有两个属性，合成属性 val 以及 val 依赖的继承属性 $base$ 。其次， $base$ 属性是从 $based-num$ 左边的子孙向右边的子孙继承的(即从 $basechar$ 到 num)。因此，在这种情况下，我们必须从右向左而不是从左向右给 $based-num$ 的子孙赋值。我们继续给出 $EvalWithBase$ 程序的伪代码用以计算 $base$ 和 val 。对于这种情况，在一遍中 $base$ 用前序遍历计算，而 val 则用后序遍历计算(后面即将讨论多属性和多遍问题)。这段伪代码如下(参见表6-4的属性文法)：

```

procedure EvalWithBase (T: treenode);
begin
  case nodekind of T of
    based-num:
      EvalWithBase ( right child of T );
      assign base of right child of T to base of left child;
      EvalWithBase ( left child of T );
      assign val of left child of T to T.val;
    num:
      assign T.base to base of left child of T;
      EvalWithBase ( left child of T );
    if right child of T is not nil then
      assign T.base to base of right child of T;
      EvalWithBase ( right child of T );
    if vals of left and right children error then
      T.val := T.base* ( val of left child ) + val of right child;
    else T.val := error;
    else T.val := val of left child;
    basechar:
      if child of T = 0 then T.base := 8
      else T.base := 10;
    digit:
      if T.base := 8 and child of T = 8 or 9 then T.val := error
      else T.val := numval ( child of T );
    end case;
end EvalWithBase;

```

我们把相应的语法树C语言声明的构造和将 $EvalWithBase$ 伪代码转换为C语言代码的工作留作练习。

在组合合成和继承属性的属性文法中，如果合成属性依赖于继承属性(及其他合成属性)，但继承属性不依赖于任何合成属性，那么就可能在分析树或语法树的一遍遍历中计算所有的属性。上例就是这一点很好的一个例子，赋值顺序可以组合 $PostEval$ 和 $PreEval$ 伪代码程序进行概括：

```

procedure CombinedEval (T: treenode );
begin
  for each child C of T do
    compute all inherited attributes of C;

```

```
CombinedEval ( C );
compute all synthesized attributes of T;
end;
```

继承属性依赖于合成属性的情形更加复杂，需要对分析树或语法树进行多遍遍历，如下一个例子所示。

例6.14 考虑下列表达式文法的简化版本：

```
exp    exp / exp | num | num.num
```

这个文法只有一个操作符——除，用记号 / 表示。它还有两种数的形式，整型数由数字序列组成，用记号 *num* 表示，浮点数用记号 *num.num* 表示。这个文法的思想是：根据操作数是浮点数还是整型数，操作符可能会有不同的解释。特别对于除法，根据是否允许有小数部分而结果完全不同。如果不允许，除法通常称作 *div* 操作，5 / 4 的值就是 5 *div* 4 = 1。如果是浮点数除法，5 / 4 的值就是 1.2。

现在假设一种编程语言要求混合表达式提供浮点计算能力，根据语义使用相应的操作。因此，表达式 5 / 2 / 2.0 (假定除法是左结合) 的含义是 1.25，而 5 / 2 / 2 的含义是 1[⊖]。描述这些语义需要 3 个属性：一个合成的布尔值属性 *isFloat*，指示表达式的任何部分是否有浮点数值；一个继承属性 *etype*，它有两个值 *int* 和 *float*，它们给出每个子表达式的类型以及哪一个表达式依赖 *isFloat*；最后是每个子表达式计算的 *val*，它依赖继承属性 *etype*。这个情况也要求对顶层表达式进行区分(这样我们知道没有更多的子表达式需要考虑)。我们给文法增加一个开始符号：

```
S    exp
```

属性等式在表 6-7 中给出。在文法规则 *exp* *num* 的等式中，我们使用 *Float* (*num.val*) 表示将整型值 *num.val* 转换成浮点数值值的函数。我们还使用 “ / ” 表示浮点数除法，使用 “ *div* ” 表示整数除法。

在这个例子中，属性 *isFloat*、*etype* 和 *val* 可以用对分析树或语法树的两遍遍历来计算。第一遍通过后序遍历计算合成属性 *isFloat*。第二遍用前序和后序遍历的组合计算继承属性 *etype* 和合成属性 *val*。我们把这些遍的描述、表达式相应的属性等式以及在语法树上进行后续遍的伪代码或 C 语言代码构造留作练习。

表6-7 例6.14的属性文法

文法规则	语义规则
S <i>exp</i>	<i>exp.etype</i> = if <i>exp.isFloat</i> then <i>float</i> else <i>int</i> <i>S.val</i> = <i>exp.val</i>
<i>exp</i> ₁ <i>exp</i> ₂ / <i>exp</i> ₃	<i>exp</i> ₁ . <i>isFloat</i> = <i>exp</i> ₂ . <i>isFloat</i> or <i>exp</i> ₃ . <i>isFloat</i> <i>exp</i> ₂ . <i>etype</i> = <i>exp</i> ₁ . <i>etype</i> <i>exp</i> ₃ . <i>etype</i> = <i>exp</i> ₁ . <i>etype</i> <i>exp</i> ₁ . <i>val</i> = if <i>exp</i> ₁ . <i>etype</i> = <i>int</i> then <i>exp</i> ₂ . <i>val</i> <i>div</i> <i>exp</i> ₃ . <i>val</i> else <i>exp</i> ₂ . <i>val</i> / <i>exp</i> ₃ . <i>val</i>

⊖ 这个规则与 C 语言中使用的规则不同。例如，在 C 语言中，5 / 2 / 2.0 的值是 1.0，而不是 1.25。

(续)

文法规则	语义规则
$exp \rightarrow num$	$exp.isFloat = \text{false}$ $exp.val =$ if $exp.etype = int$ then $num.val$ else $Float(num.val)$
$exp \rightarrow num.num$	$exp.isFloat = \text{true}$ $exp.val = num.num.val$

6.2.3 作为参数和返回值的属性

通常在计算属性时，利用参数和返回的函数值与属性值进行通信，而不是把它们作为字段存储在语法树的记录结构中，这样做是有意义的。当许多属性值都相同，或者仅仅在计算其他属性值时临时使用，就尤为重要。对于这种情况，使用语法树的空间在每个节点存储属性值就没什么意义了。事实上，递归遍历程序用前序计算继承属性，而用后序计算合成属性，在子节点把继承属性作为参数传递给递归函数调用，并接收合成属性作为那些相同调用的返回值。前几章已经出现了几个这样的例子。特别地，算术表达式合成属性值的计算能通过递归分析程序返回当前表达式的值进行。类似地，在语法分析期间，因为直到它的构造完成，也没有用以记录作为属性的自身的数据结构存在，所以语法树自身作为合成属性必须通过返回值计算。

对于更复杂的情况，例如当要返回不止一个合成属性时，就有必要使用记录结构或联合作为返回值，或者针对不同的情况把递归程序分割成几个程序。我们用一个例子来说明这一点。

例6.15 考虑例6.13中的递归程序 *EvalWithBase*。在这个程序中，一个数的 *base* 属性只计算一次，之后就用于随后的 *val* 属性的计算。类似地，在一个完整的数的值的计算中，数的部分 *val* 属性只是临时使用。把 *base* 转换成参数 (作为继承属性) 和把 *val* 转换成返回值是有意义的。将 *EvalWithBase* 程序修改如下：

```

function EvalWithBase (T: treenode; base: integer): integer;
var temp, temp2: integer;
begin
  case nodekind of T of
    based-num:
      temp := EvalWithBase ( right child of T );
      return EvalWithBase ( left child of T, temp );
    num:
      temp := EvalWithBase ( left child of T, base );
      if right child of T is not nil then
        temp2 := EvalWithBase ( right child of T, base );
        if temp error and temp2 error then
          return base*temp + temp2
        else return error;
      else return temp;

```

```

basechar:
    if child of T = 0 then return 8
    else return 10;
digit:
    if base = 8 and child of T = 8 or 9 then return error
    else return numval (child of T);
end case;
end EvalWithBase;

```

当然，只有工作在 *base* 属性和 *val* 属性才有相同的 *integer* 数据类型，因为在一种情况下 *EvalWithBase* 返回 *base* 属性(当分析树节点是一个 *basechar* 节点)，在另一种情况下 *EvalWithBase* 返回 *val* 属性。在第1次调用 *EvalWithBase* 时也有些不规则(在分析树根节点 *base-num* 节点)，即使它可能不存在，随后再忽略掉，此时也必须提供一个 *base* 值。例如，启动计算时，必须进行这样的调用

EvalWithBase (rootnode , 0) ;

哑元的基值为0。因此，区分3种情况：*base_num*、*basechar*和*digit*，并且对这3种情况写出3个独立的程序是合理的。伪代码如下：

```

function EvalBasedNum (T: treenode) : integer;
(*only called on root node*)
begin
    return EvalNum ( left child of T, EvalBase (right child of T) );
end EvalBasedNum;

function EvalBase ( T: treenode ): integer;
(*only called on basechar node*)
begin
    if child of T = 0 then return 8
    else return 10;
end EvalBase;

function EvalNum ( T: treenode; base: integer ): integer;
var temp, temp2: integer;
begin
    case nodekind of T of
        num:
            temp := EvalWithBase ( left child of T, base );
            if right child of T is not nil then
                temp2 := EvalWithBase ( right child of T, base );
                if temp = error and temp2 = error then
                    return base*temp + temp2
                else return error;
            else return temp;

```



```

digit:
    if base = 8 and child of T = 8 or 9 then return error
    else return numval ( child of T );
end case;
end EvalNum;

```

6.2.4 使用扩展数据结构存储属性值

对那些不能方便地把属性值作为参数或返回值使用的情况（特别是当属性值有重要的结构时，在翻译时可能专门需要），把属性值存储在语法树节点也是不合理的。在这些情况中，如查表、图以及其他一些数据结构对于获得属性值的正确活动和可用性都是有用的。通过由到表示用于维护属性值的相应的数据结构调用替换属性等式（表示属性值的赋值）就可反映出属性文法自身可以被修改的这个需要。这导致语义规则不再表示一个属性文法，但在描述属性的语义时仍然有用，而程序的操作清晰了。

例6.16 考虑前一个例子使用参数和返回值的 *EvalWithBase* 程序。因为属性一旦设置，在值计算过程中就固定了，我们可以使用一个非局部变量存储它的值，而不是在每次作为一个参数传递（如果 *base* 不固定，这样一个递归过程是危险的甚至是不正确的）。因此，我们可以改变 *EvalWithBase* 的伪代码如下：

```

function EvalWithBase (T: treenode): integer;
var temp, temp2: integer;
begin
    case nodekind of T of
        based-num:
            SetBase ( right child of T );
            return EvalWithBase ( left child of T );
        num:
            temp := EvalWithBase ( left child of T );
            if right child of T is not nil then
                temp2 := EvalWithBase ( right child of T );
                if temp = error and temp2 = error then
                    return base*temp + temp2
                else return error;
            else return temp;
    end case;
    digit:
        if base = 8 and child of T = 8 or 9 then return error
        else return numval ( child of T );
    end case;
end EvalWithBase;

procedure SetBase (T: treenode);
begin
    if child of T = 0 then base := 8

```

```
else base := 10;
end SetBase;
```

这里我们把向非局部变量 *base* 赋值的过程分离到 *SetBase* 程序中，它只在 *basechar* 节点上调用。*EvalWithBase* 剩下的代码则只是直接引用 *base*，而不是作为一个参数传递。

也可以改变语义规则来反映非局部变量 *base* 的使用。在这种情况下，规则就像下面的一样使用赋值明确地指示非局部变量 *base*：

文法规则	语义规则
<i>based-num</i>	
<i>num basechar</i>	<i>based-num.val</i> = <i>num.val</i>
<i>basechar</i> o	<i>base</i> := 8
<i>basechar</i> d	<i>base</i> := 10
<i>num</i> ₁ <i>num</i> ₂ <i>digit</i>	<i>num</i> ₁ . <i>val</i> = if <i>digit.val</i> = <i>error</i> or <i>num</i> ₂ . <i>val</i> = <i>error</i> then error else <i>num</i> ₂ . <i>val</i> * <i>base</i> + <i>digit.val</i>
etc.	etc.

在这个意义上，*base* 现在不再是一个到目前为止所使用的属性，语义规则也不再构成一个属性文法。然而，如果将 *base* 看作是具有相关特性的变量，这些规则就仍然为编译器编写者充分定义了 *base-num* 的语义。

对于语法树，外部数据结构最初的一个例子是符号表 (symbol table)，结合程序中声明的常量、变量和过程存储属性。符号表是一种目录数据结构，有 *insert*、*lookup*、*delete* 这样的操作。下一节讨论在一个典型的程序设计语言中符号表的问题。这一节介绍下面这个简单的例子。

例6.17 考虑表6-3中简单声明的属性文法，这个属性文法的属性赋值程序在例 6.12 给出。一般地，声明中信息使用声明标识符作为关键字插入到符号表中并存储在哪里，以供之后程序的其他部分翻译使用。因此，假设对这个符号表所在的文法通过 *insert* 程序而把标识符名、它声明的数据类型和名数据类型插入到符号表中。声明如下：

```
procedure insert (name : string ; dtype : typekind);
```

因此替代在语法树中存储每个变量的数据类型，使用这个程序把它插入到符号表中。而且因为每个声明只有一个相关类型，所以就可在处理过程中使用一个全局变量存储每个声明的常量 *dtype*。结果的语义规则如下：

文法规则	语义规则
<i>decl</i> <i>type var-list</i>	
<i>type</i> int	<i>dtype</i> = <i>integer</i>
<i>type</i> float	<i>dtype</i> = <i>real</i>
<i>var-list</i> ₁ id , <i>var-list</i> ₂	<i>insert</i> (<i>id.name</i> , <i>dtype</i>)
<i>var-list</i> id	<i>insert</i> (<i>id.name</i> , <i>dtype</i>)

在对 *insert* 的调用中，我们使用 *id.name* 查阅标识符字符串，即假定由扫描器或分析程序

要计算的。这些语义规则与相应的属性文法完全不同；事实上，对于文法规则 *decl* 就完全没有语义规则。因为对 *insert* 的调用依赖于在 *type* 规则中设置的 *dtype*，所以虽然很清楚在相关的 *var-list* 规则之前必须处理 *type* 规则，但依赖仍不像表达的那样清晰。

相应的属性赋值程序 *EvalType* 的伪代码如下 (与例6-12的代码相比较)：

```

procedure EvalType ( T: tree node );
begin
  case nodekind of T of
    decl:
      EvalType ( type child of T );
      EvalType ( var-list child of T );
    type:
      if child of T = int then dtype := integer;
      else dtype := real;
    var-list:
      insert ( name of first child of T , dtype )
      if third child of T is not nil then
        EvalType ( third child of T );
      end case;
end EvalType;

```

6.2.5 语法分析时属性的计算

有一个很自然会出现的问题，即在分析阶段的同时要计算哪种扩展属性，而无须等到通过语法树的递归遍历进行对源代码的多遍处理。这对于语法树自身特别重要，如果合成属性要被后面的语义分析使用，它必须在语法分析期间构造。在历史上，因为注意的重点是编译器进行一遍翻译的能力，所以对语法分析阶段计算所有属性的可能性产生了很大的兴趣。现在这一点已不太重要了，因此我们没有对所有已经开发的特定技术提供详尽的分析。然而，这个思想和要求总的来看是有价值的。

在一次分析中哪些属性能成功地计算在很大程度上要取决于使用分析方法的能力和性质。这样一个重要的限制是所有主要的分析方法都从左向右处理输入程序（这也是前面两章研究 LL 和 LR 分析技术的第 1 个 L 的内容）。它等价于要求属性能通过从左向右遍历分析树进行赋值。对于合成属性这不是一个限制，因为节点的子节点可以用任意顺序赋值，特别是从左向右。但是对于继承属性，这就意味着在相关图中没有“向后”的依赖（在分析树中依赖从右指向左）。例如，例 6.4 的属性文法违反了这个特性，因为 *based-num* 通过后缀 *o* 或 *d* 给出其基于的进制，在字符串结尾看到后缀并进行处理之前，*val* 属性都不能进行计算。属性文法确保的这个特性称作 L-属性 (从左向右)，我们给出以下定义。

定义 属性 a_1, \dots, a_k 的一个属性文法是 L-属性 (L-attributed)，如果对每个继承属性 a_j 和每个文法规则

$$X_0 \rightarrow X_1 X_2 \dots X_n$$

a_j 的相关等式都有以下形式

$$X_i.a_j = f_{ij}(X_0.a_1, \dots, X_0.a_k, X_1.a_1, \dots, X_1.a_k, \dots, X_{i-1}.a_1, \dots, X_{i-1}.a_k)$$

也就是说, 在 X_i 处 a_j 的值只依赖于在文法规则中 X_i 左边出现的符号 X_0, \dots, X_{i-1} 的属性。

作为一个特例, 我们已经注意到 S- 属性文法是 L- 属性文法。

给定一个 L- 属性文法, 其继承属性不依赖于合成属性, 如前所述, 通过把继承属性转换成参数以及把合成属性转换成返回值, 递归下降的分析程序可以对所有的属性赋值。然而, LR 分析程序, 如 Yacc 产生的 LALR(1) 分析程序, 适合于处理主要的合成属性。反过来说, 其原因在于 LR 分析程序的功能强于 LL 分析程序。当已知在一个派生中使用的文法规则时, 属性才变成可计算的, 这是因为只有在那时才能确定属性计算的等式。但是, LR 分析程序将确定在派生中使用的文法规则推迟到文法规则的右部完全形成时才使用。这使得使用继承属性十分困难, 除非它们的特性对于所有可能的右部选择都是固定的。我们将简要地讨论在 Yacc 应用中最通常的情况下使用分析栈来计算属性。更复杂的技术将在“注意与参考”一节讲述。

1) LR 分析中合成属性的计算 对于 LR 分析程序而言这是一种简单的情况。LR 分析程序中通常由一个值栈存储合成属性(如果对每个文法符号有不只一个属性, 可能是联合或结构)。值栈将和分析栈并行操作, 根据属性等式每次在分析栈出现移进或规约来计算新值。我们用表 6-8 来说明这一点, 属性文法在表 6-5 中, 这是简单算术表达式的二义性版本。为简单起见, 我们对文法使用缩写符号, 并且忽略了表中 LR 分析算法的一些细节。特别地, 没有指明状态号、没有显示增加的开始符号, 也没有表达隐含的二义性消除规则。这个表除了通常的分析动作之外, 还有两个新栏: 值栈和语义动作。语义动作指示当分析栈出现规约时值栈发生的计算(移进看作是把记号值同时推进分析栈和值栈, 因此这和单独的分析程序不同)。

作为语义动作的例子, 考虑表 6-8 的第 10 步。值栈包含整数 12 和 5, 由记号 + 分割开, 5 在值栈的栈顶。分析动作通过 $E \rightarrow E + E$ 规约, 根据表 6-5, 相应的语义动作是按照等式 $E_1.val = E_2.val + E_3.val$ 计算。在栈顶分析程序相应的动作如下(伪代码):

```
pop t3      {从值栈中取出  $E_3.val$ }
pop         {丢弃 + 记号}
pop t2      {从值栈中取出  $E_2.val$ }
t1 = t2 + t3 {加}
push t1     {将结果压进值栈}
```

在 Yacc 中第 10 步的规约表示的情形可以写成如下规则:

```
E : E + E { $$ = $1 + $3 }
```

这里伪变量 \$i 表示规则右部的内容被规约, 并通过从右部向后计数转换成值栈的内容。因此, 可以在栈顶找到对应于最右端的 E 的 \$3, 而在栈顶下面的两个位置找到 \$1。

表 6-8 在 LR 分析中表达式 $3*4+5$ 的语法和语义动作

	分析栈	输入	语法动作	值栈	语义动作
1	\$	3*4+5 \$	移进	\$	
2	\$ n	*4+5 \$	规约 $E \rightarrow n$	\$ n	$E.val = n.val$
3	\$ E	*4+5 \$	移进	\$ 3	
4	\$ E *	4+5 \$	移进	\$ 3 *	
5	\$ E * n	+5 \$	规约 $E \rightarrow n$	\$ 3 * n	$E.val = n.val$

(续)

	分析栈	输入	语法动作	值栈	语义动作
6	\$ E * E	+5 \$	规约E E * E	\$ 3 * 4	$E_1.val = E_2.val * E_3.val$
7	\$ E	+5 \$	移进	\$ 12	
8	\$ E +	5 \$	移进	\$ 12 +	
9	\$ E + n	\$	规约E n	\$ 12 + n	$E.val = n\ val$
10	\$ E + E	\$	规约E E + E	\$ 12 + 5	$E_1.val = E_2.val + E_3.val$
11	\$ E	\$		\$ 17	

2) 在LR分析中继承前面计算的合成属性 因为LR分析使用从左向右的赋值策略, 因为这些值已经被压进了值栈, 所以与规则右边非终止符相关的动作可以把符号的合成属性使用到规则的左边。为了简要说明这一点, 可考虑产生式选择 $A \rightarrow B C$, 假设 C 有一个继承属性 I 和以某种方式依赖于 B : $C.i = f(B.s)$ 的合成属性 s 。通过在 B 和 C 之间引入一个 ϵ -产生式安排值栈栈顶的存储, 在识别 C 之前的 $C.i$ 值可以存进一个变量中:

文法规则	语义规则
$A \rightarrow B C D$	
$B \rightarrow \dots$	{计算 $B.s$ }
$D \rightarrow \epsilon$	$saved_i = f(valstack[top])$
$C \rightarrow \dots$	{现在 $saved_i$ 是可用的}

Yacc中的这个处理十分容易, 因为无须明确地引入 ϵ -产生式。只需在调度的规则处写入存储计算属性的动作就行了:

```
A : B { saved_i = f($1); } C ;
```

(这里伪变量 $\$1$ 指的是 B 的值, 动作执行时它在栈顶)。Yacc中这样的嵌入动作在第4章5.5.6节中已讲述过。

当总能预测出过去计算的合成属性的位置时, 可以使用这个策略的另一方面。对于这种情况, 无须将值拷贝到变量中, 在值栈中能够直接访问。例如, 考虑下列具有继承的 $dtype$ 属性的L-属性文法:

文法规则	语义规则
$decl \rightarrow type\ var_list$	$var_list.dtype = type.dtype$
$type \rightarrow int$	$type.dtype = integer$
$type \rightarrow float$	$type.dtype = real$
$var_list_1 \rightarrow var_list_2, id$	$insert(id.name, var_list_1.dtype)$ $var_list_2.dtype = var_list_1.dtype$
$var_list \rightarrow id$	$insert(id.name, var_list.dtype)$

在这种情况下, 在第1个 var_list 识别之前可以将 $dtype$ 属性作为非终结符 $type$ 的合成属性计算进入到值栈中。然后当 var_list 的每条规则规约时, 在值栈中通过从栈顶向后计数就可以在固定位置找到 $dtype$ 。当 $var_list \rightarrow id$ 规约时, $dtype$ 就在栈顶的下面, 而当 $var_list_1 \rightarrow var_list_2, id$ 规约时, $dtype$ 在栈顶下面的3个位置。通过上面的属性文法中为 $dtype$ 消除两个拷贝规则并直接访问值栈, 可以实现这个LR分析程序。

文法规则	语义规则
<i>decl</i> <i>type var-list</i>	
<i>type</i> int	<i>type.dtype</i> = integer
<i>type</i> float	<i>type.dtype</i> = real
<i>var-list</i> ₁ <i>var-list</i> ₂ , id	insert (<i>id.name</i> , <i>valstack</i> [<i>top</i> - 3])
<i>var-list</i> id	insert (<i>id.name</i> , <i>valstack</i> [<i>top</i> - 1])

(注意：因为 *var-list* 没有合成属性 *dtype*，分析程序必须在栈中压入一个虚值以保持栈中的正确位置)。

这种方法存在几个问题。首先，它要求程序员在分析过程中直接访问值栈，这在自动产生分析程序时是有风险的。例如，在当前的规则被认可之下 Yacc 没有像上面的方法所要求的访问值栈的伪变量转换。因此，要在 Yacc 中实现这一方案就必须编写特定的代码。第 2 个问题是这个技术仅使用在前面计算的属性的位置能从文法中推断出来的情况下。例如，我们编写了上面的声明文法，*var-list* 是右递归的(就像例 6.17 中的)，然后在栈中有 *id* 的一个仲裁数，而 *dtype* 在栈中的位置是未知的。

到目前为止，在 LR 分析中处理继承属性最好的技术是使用外部数据结构，如符号表或非局部变量，保存继承属性值，并增加 ϵ -产生式(或像在 Yacc 中的嵌入动作)，考虑在适当的时刻这些数据结构的变化。例如，刚讨论的 *dtype* 问题的一种解决办法可以从对 Yacc 的嵌入动作的讨论中得到(见 5.5.6 节)。

我们要认识到，即使后一种方法没有排除陷阱，在文法中增加 ϵ -产生式也可能会增加分析冲突，因此对任意的变量 *k*，LALR(1) 文法均能转变成非 LR(*k*) 文法(见练习 6.15 以及“注意与参考”一节)。在实际情况中，这很少发生。

6.2.6 语法中属性计算的相关性

作为本节最后一个主题，值得注意的是属性严重依赖于文法结构的特性。可能会有这样的情况，不改变语言合法的字符串而修改文法会使属性的计算更简单或更复杂。当然，有以下定理：

定理：(Knuth [1968]) 给定一个属性文法，通过适当地修改文法，而无须改变文法的语言，所有的继承属性可以改变成合成属性。

我们给出一个例子，说明如何通过修改文法继承属性转换成合成属性。

例 6.18 考虑前一个例子中简单声明的文法：

```
decl    type var-list
type    int | float
var-list id, var-list | id
```

表 6-3 的属性文法的 *dtype* 属性是继承属性。然而，如果重写这个文法如下：

```
decl    var-list id
var-list var-list id, | type
type    int | float
```


那么产生了相同的字符串，但根据下面的属性文法，属性 *dtype* 现在变成了合成属性：

文法规则	语义规则
<i>decl</i> <i>var-list id</i>	<i>id.dtype</i> = <i>var-list.dtype</i>
<i>var-list</i> ₁ <i>var-list</i> ₂ <i>id</i> ,	<i>id.dtype</i> = <i>var-list</i> ₂ . <i>dtype</i>
<i>var-list</i> <i>type</i>	<i>var-list</i> ₁ . <i>dtype</i> = <i>var-list</i> ₂ . <i>dtype</i>
<i>type</i> int	<i>type.dtype</i> = <i>integer</i>
<i>type</i> float	<i>type.dtype</i> = <i>real</i>

我们在图6-10中已说明了文法的这个改变对语法树和 *dtype* 属性计算的影响怎样，它显示了字符串 **float x,y** 语法树及其属性值和依赖。在图中父节点或兄弟节点值的两个 *id.dtype* 值的依赖用虚线画出。而这些依赖的出现违反了在这个属性文法中没有继承属性的要求，事实上这些依赖总是留在语法树中(即是非递归的)，并且可能由相应的父节点的操作实现。因此，这些操作不看成是继承的。

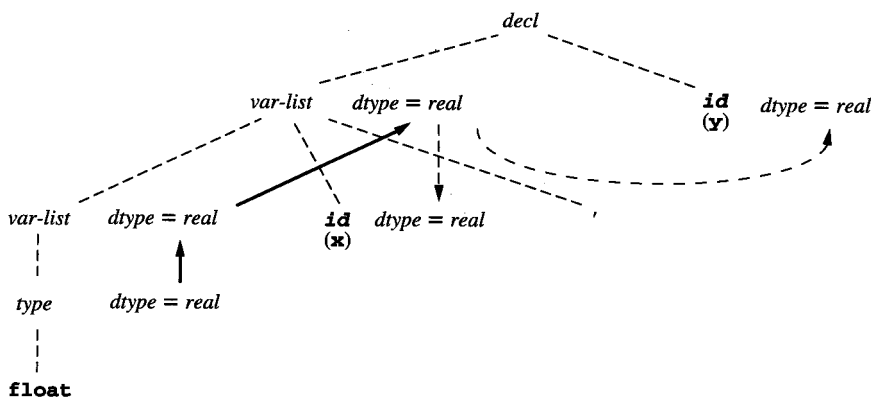


图6-10 字符串 **float x,y** 的语法树，显示例6.18中属性文法指定的 *dtype* 属性

事实上，状态理论并不像它看起来那样有用。为了把继承属性转换成合成属性而对文法进行改变通常会使文法和语义规则更加复杂和难以理解。因此，我们不推荐使用这种方法来处理计算继承属性的问题。另一方面，如果一个属性计算看起来非常困难，可能是因为这个文法用了不适合于计算它的方法来定义，也就值得对这个文法进行修改。

6.3 符号表

符号表是编译器中的主要继承属性，并且在语法树之后，形成了主要的数据结构。虽然因为一些例外我们推迟了对符号表的讨论，但它仍对语义分析阶段概念上的框架最适合，读者也会意识到在实际的编译器中，符号表通常紧密地和语法分析程序甚至扫描器相关，它们可能需要直接向符号表中输入信息，或者通过它消除二义性（C语言中一个这样的例子可参见练习6.22）。然而，在一种设计非常仔细的语言（如Ada或Pascal）中，有可能甚至有理由将符号表的操作推迟到一个完整的阶段之后，这时在语法上已知要翻译的程序是正确的。例如，在TINY编译器中就这样做了，其符号表将在本章的后面讨论。

符号表主要的操作有插入、查找和删除；其他的一些操作也是必要的。当处理新定义的名

字时,插入操作依赖存储这些名字定义所提供的信息。当相关的代码使用名字时,查找操作找出与名字对应的信息。而当不再运用名字的定义时,需要用删除操作除去名字定义提供的信息。⊙ 这些操作的特性由被翻译的编程语言的规则指定。特别地,在符号表中需要存储什么信息是结构的功能和名字定义的目的。一般包括数据类型信息、应用区域信息(作用域,下面将讨论)以及在存储器中的最终定位信息。

这一节我们将首先讨论符号表数据结构的组织,以便快速而方便地进行访问。随后将描述一些典型语言的要求和它们在符号表操作上的影响。最后,给出了一个使用属性文法符号表的例子。

6.3.1 符号表的结构

在编译器中符号表是一个典型的目录数据结构。插入、查找和删除这3种基本操作的效率根据数据结构的组织的不同而变化很大。对不同组织结构效率的分析以及对好的组织策略的研究是数据结构课程的一个主要主题。因此,本书对这个题目不详细讨论,但我们提请读者参考本章最后的“注意与参考”一节提及的资料,获取更多的信息。这里,我们将给出编译器构造中这些表格最有用的数据结构的概要。

目录结构的典型实现包括线性表、各种搜索树结构(二叉搜索树、AVL树、B树)以及杂凑表(hash表)等。线性表是一种较好的基本数据结构,它能够提供最方便而直接的实现,即用恒定次数的插入操作(通常插入在前面或后面)以及查找和删除操作,表的大小是线性的。这对编译器的实现是很好的,不必去关心编译的速度,就像一个原型或实验编译器,或者用于非常小的程序的解释器。搜索树结构对符号表的用处稍微小一点,部分是因为它们没有提供最好的效率,也因为删除操作的复杂性。杂凑表通常为符号表的实现提供了最好的选择,因为所有3种操作都能在几乎恒定的时间内完成,在实践中也最常使用。因此,对杂凑表的讨论更加详细一些。

杂凑表是一个入口数组,称作“桶(bucket)”,使用一个整数范围的索引,通常从0到表的尺寸减1。杂凑函数(hash function)把索引键(在这种情况下是标识符名,组成一个字符串)转换成索引范围内的一个整数的杂凑值,对应于索引键的项存储在这个索引的“桶”中。必须非常小心,杂凑函数在索引范围内尽可能一致地分配键索引,因为杂凑冲突(collision)(两个键由杂凑函数映射到相同的索引)在查找和删除操作时将引起性能的下降。杂凑函数也需要在一个恒定的时间内操作,或者至少根据键的尺寸时间是线性的(如果键的尺寸有界,这可以计算恒定时间)。我们不久将研究杂凑函数。

一个重要的问题是杂凑表如何处理冲突(这称为冲突解决(collision resolution))。一种方法是在每个“桶”中为一个项分配刚好够的空间,通过在连续的“桶”中插入新项来解决冲突(这有时称作开放寻址(open addressing))。在这种情况下,杂凑表的内容由表所使用的数组的大小限制,当数组填写冲突越来越频繁时,就会引起性能的显著下降。这个方法进一步的问题是,至少对编译器的结构而言实现删除操作比较困难,并且删除不会改进后继表的性能。

编译器结构最好的方案对应于开放寻址也许是另一种方法,称作分离链表(separate chaining)。在这种方法中每个“桶”实际上是一个线性表,通过把新的项插入到“桶”表中来解决冲突。图6-11给出了这种方案的一个简单的例子,其杂凑表的尺寸是5(小得很不现实,仅作演示用)。在那个表中,假定插入了4个标识符(i、j、size和temp),并且size和j有相同

⊙ 与销毁这个信息相比,删除操作更像是从可视区移走,可能是存储到别处,或者把它标记成不活动的。

的杂凑值(命名为1)。在图中我们看到,“桶”号为1的表中size在j的前面;表中项的顺序依赖于插入的顺序以及表维护的方式。一种通常的方法是总是插入在表的开始,这样使用这种方法size在j之后插入。

图6-11也显示了在每个“桶”中作为链接列表实现的列表(实心圆点用来表示空指针)。

这些可以使用编译器实现语言的动态指针分配方法进行分配,或者在编译器自身的空间数组中手工分配。

编译器编写者必须回答的一个问题是要初始化多大的“桶”数组。通常,在编译器构成时,这个大小就固定了。⊙一般的大小范围从几百到上千。如果动态分配实际的入口,即使很小的数组也允许编译很大的程序,只是花费一些额外的编译时间。在任何情况下,“桶”数组的实际大小要选择一个素数,因为这将使一般的杂凑函数运行得更好。例如,如果希望“桶”数组的大小是200,就应该选择211作为数组的大小而不是200(211是大于200的最小的素数)。

现在我们转向描述通用的杂凑函数。在符号表实现中使用的杂凑函数将字符串(标识符名)转换成 $0 \dots size-1$ 范围内的一个整数。一般这通过3步来进行。首先,字符串中的每个字符转换成一个非负整数。然后,这些整数用一定的方法组合形成一个整数。最后,把结果整数调整到 $0 \dots size-1$ 范围内。

通常使用编译器实现语言内嵌的转换机构把每个字符转换成非负整数。例如, Pascal的ord函数将字符串转换成整数,通常是其ASCII值。类似地,在C语言中,如果要在算术表达式中使用字符或把它赋给一整型变量,就自动地将其转换成整数。

使用数学上的取模(modulo)函数很容易调整一个非负整数落到 $0 \dots size-1$ 范围内,它返回用size去除一个数所得的余数。这个函数在Pascal中称作mod,在C中用%表示。使用这种方法时,size是一个素数,这一点很重要。否则,随机分配的整数集不会将标定的值随机分配到 $0 \dots size-1$ 范围内。

留给杂凑表实现者选择一个方法,把字符的不同整数值组合成一个非负整数。一种简单的方法是忽略许多字符,只把开头的几个字符,或第一个、中间的和最后一个字符的值加在一起。这对编译器来说是不适当的,因为编程者倾向于成组分配变量名,像temp1、temp2,或m1tmp、m2tmp等等,并且这种方法在这样的名字中会经常引起冲突。因此,选择的方法要包括每个名字中所有的字符。另一种流行但不适当的方法是简单地把所有字符的值加起来。使用那种方法,所有排列相同的字符,像tempx和xtemp,会引起冲突。

这些问题的一个好的解决办法是,当加上下一个字符的值时,重复地使用一个常量作为乘法因子。因此,如果 c_i 是第 i 个字符的数字值, h_i 是在第 i 步计算的部分杂凑值,那么 h_i 根据下面的递归公式计算, $h_0 = 0$, $h_{i+1} = h_i \cdot \alpha + c_{i+1}$,最后的杂凑值用 $h = h_n \bmod size$ 计算。这里 n 是杂凑的名字中字符的个数。这等价于下列公式

$$h = (\alpha^{n-1}c_1 + \alpha^{n-2}c_2 + \dots + \alpha c_{n-1} + c_n) \bmod size = \left(\sum_{i=1}^n \alpha^{n-i} c_i \right) \bmod size$$

当然,在这个公式中 α 的选择对输出结果有重要影响。 α 的一种合理的选择是2的幂,如16

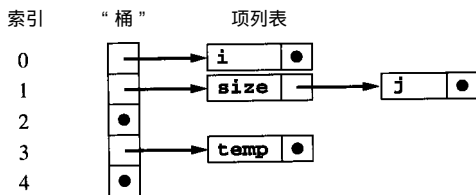


图6-11 分离链接的杂凑表,显示如何解决冲突

⊙ 如果杂凑表增长得太大,在不工作时可有方法增大数组的尺寸(和改变杂凑函数),但这很复杂也很少使用。

或128, 这样乘法可以通过移位来完成。实际上, 选择 $m=128$ 的结果是把字符串看成是基于128的数, 假定字符值 c_i 都小于128 (对ASCII字符为真)。在本文中提到的其他可能性是各种素数 (见“注意与参考”一节)。

在公式中 h 的溢出有时也成问题, 特别是在双字节整数的机器中对于较大的 m 值。如果整数值用于计算, 溢出将导致负数 (在双字节补码表示中), 并引起执行错误。在这种情况下, 通过在求和循环中执行 **mod** 操作可以得到相同的结果。在程序清单 6-2 中给出了杂凑函数 h 的C代码的例子, 使用前面的公式, m 的值为16 (进行4次移位, 因为 $16 = 2^4$)。

程序清单6-2 符号表的杂凑函数

```
#define SIZE ...
#define SHIFT 4

int hash ( char * key )
{ int temp = 0;
  int i = 0;
  while (key[i] != '\0')
  { temp = ((temp << SHIFT) + key[i]) % SIZE;
    ++i;
  }
  return temp;
}
```

6.3.2 说明

符号表的行为严重依赖于要翻译的语言的特性。例如, 当需要调用插入和删除操作时, 它们如何作用于符号表, 以及什么属性插入到表中, 不同的语言都有很大的变化。甚至当符号表建立时翻译/执行过程的时间和符号表需要存在多长时间, 对不同的语言也完全不同。这一节我们简要说明几种语言中影响符号表行为和实现的有关声明的问题。

在编程语言中经常出现的4种基本说明是: 常量说明、类型说明、变量说明和过程函数说明。

常量声明(constant declaration)包括C语言中的**const**声明, 如

```
const int SIZE = 199;
```

(C语言还有一个**#define**机构来创建常量, 但那是在预处理时进行的, 而不是严格意义上的编译器处理阶段)。

类型声明(type declaration)包括Pascal语言中的类型声明, 如

```
type Table = array [1..SIZE] of Entry;
```

以及C语言中的**struct**和**union**说明, 如

```
struct Entry
{ char * name;
  int count;
  struct Entry * next;
};
```

这里用名字**Entry**说明了一个结构类型。C语言中还有一个**typedef**机构用来说明类型的别名

```
typedef struct Entry * EntryPtr;
```

变量说明(variable declarations)是说明中最常用的形式, 包括FORTRAN语言中的说明, 如

```
integer a,b(100)
```

以及C语言中的说明,如

```
int a,b[100];
```

最后,是过程/函数说明(procedure/function declarations),如程序清单6-2中说明的C函数。这实际上不比过程/函数类型的常数说明多什么东西,但因为它们特别的特性,通常从语言说明中分离出来。这些说明是明确的(explicit),使用了特定的语言结构进行说明。也可能有隐含的(implicit)说明,说明依附于执行的指令而不明确说明。例如,FORTRAN和BASIC允许使用没有明确说明的变量。在这些隐含说明中,使用了一些约定提供由明确说明给出的信息。例如,FORTRAN语言有类型的约定,如果没有使用明确的说明,字母I到N开头的变量自动说明为整型,其他的是实型。隐含说明也可称作使用时说明(declaration by use),因为没有明确说明的变量在第一次使用时可看成隐含包含了其说明。

通常最容易的是用一张符号表保存所有不同种类的说明的名字,特别是当语言禁止在不同种类的说明中使用相同的名字。有时候,对每种说明使用不同的符号表也较容易,例如,所有的类型说明包含在一张符号表中,而所有的变量说明包含在另一张符号表中。对于某些语言,特别是Algol派生的语言,如C、Pascal和Ada,希望程序不同的区域(如过程)都有独立的符号表,并按照语言的语义规则链接到一起(马上我们将更详细地讨论这一点)。

根据说明的种类,限定名字的属性也不同。常量说明给名字赋一个值,因此有时把常量说明称作值约束(value binding)。被约束的值决定编译器如何处理它们。例如,Pascal和Modula-2要求常量说明的值是静态的,因此由编译器计算。在编译期间编译器可以使用符号表用值来替代常量名。其他语言,如C和Ada,允许常量是动态的,即在执行期间才计算。这样的常量处理起来更像变量,在执行期间必须产生代码来计算它们的值。然而,这样的常量是单一指派(single assignment),一旦确定了其值就不再改变。常量说明可以明确或隐含地把数据类型约束到名字。例如,在Pascal语言中,常量的数据类型根据其(静态)值隐含地确定,而在C语言中数据类型显式地给出,就像变量说明一样。

类型说明可以把名字约束为新构造的类型,也可以为存在的已命名的类型创建一个别名。类型名通常用来和类型等价算法协作,按照语言的规则完成程序的类型检查。本章后面的一节将专门讨论类型检查,这里不再进一步讨论类型说明。

变量说明最常用于给名字限定数据类型,像在C语言中

```
Table symtab;
```

通过用名字Table表示的数据类型约束名字为symtab的变量。变量说明也可以隐含地约束其他属性。其中对符号表有主要影响的一个属性是说明的作用域(scope)或说明起作用的程序的区域(也就是变量说明可到达的区域)。作用域通常由变量在程序中说明的位置包含,但也可能被隐含的动态符号影响并和其他说明相互作用。作用域也可能是常量、类型和过程说明的特性。不久我们将更详细地讨论作用域规则。

涉及作用域的变量通过说明明确或隐含地约束,其属性是为说明的变量分配内存,以及执行分配的时机(有时称作生存期(lifetime)或说明的宽度(extent))。例如,在C语言中,所有在函数外部说明的变量都静态分配(即在执行开始前进行),因此宽度等于整个程序的执行时间,而在函数内说明的变量只在每个函数调用期间才分配(因此称作自动(automatic)分配)。通过在说明中使用关键字static,C语言也允许函数内说明的宽度从自动变为静态,例如

```
int count(void)
{ static int counter = 0;
```



```
    return ++counter;
}
```

函数 `count` 有一个静态的局部变量 `counter`，每次调用都保留其值，因此 `count` 返回当前它被调用的次数。

C语言也区分说明是用来控制内存分配还是用于类型检查。在 C语言中，任何用关键字 `extern` 开头的说明不用来执行分配。因此，如果前一个函数写成

```
int count(void)
{ extern int counter;
  return ++counter;
}
```

变量 `counter` 可以在程序的任何地方分配和初始化)。C语言把分配内存的称作说明 (definitions)，而保留单词 “**declaration** (说明)” 用于不需要分配内存的说明。因此，用关键字 `extern` 开头的说明不是一个定义，但一个标准的变量说明，如

```
int x;
```

是一个定义。在C语言中，相同的变量可能有许多说明，但只有一个定义。

存储器分配策略像类型检查一样在编译器的设计中形成了一个复杂而重要的部分，它是运行时环境 (runtime environment) 结构的一部分。下一章全部研究的都是环境，因此这里就不再进一步研究分配了，而转向在符号表中作用域和维护作用域策略的分析。

6.3.3 作用域规则和块结构

编程语言中的作用域规则变化很广，但对许多语言都有几条公共的规则。在本节中，我们讨论其中两条，使用前说明和块结构的最近嵌套规则。

使用前说明 (declaration before use) 是一条公共规则，在C和Pascal中使用，要求程序文本中的名字要在对它的任何引用之前说明。使用前说明允许符号表在分析期间建立，当在代码中遇到对名字的引用时进行查找；如果查找失败，在使用之前就出现说明错误，编译器给出相应的出错消息。因此，使用前说明有助于实现一遍编译。有些语言不需要使用前说明 (Modula-2 是一个例子)，在这样的语言中需要单独的一遍来构成符号表：一遍编译是不可能的。

块结构 (block structure) 是现代语言的一个公共特性。编程语言中的一块 (block) 是能包含说明的任意构造。例如，在Pascal中，块是主程序和过程/函数说明。在C中，块是编译单元 (也就是代码文件)、过程/函数说明以及复合语句 (用花括号括起来的语句序列 { . . . })。在C中结构和联合 (Pascal中的记录) 也可看成块，因为它们包含字段说明。类似地，面向对象编程语言中的类说明是块。一种语言是块结构 (block structured) 的，如果它允许在其他块的内部嵌入块，并且如果一个块中说明的作用域限制在本块以及包含在本块的其他块中，服从最近嵌套规则 (most closely nested rule)：为同一个名字给定几个不同的说明，被引用的说明是最接近引用的那个嵌套块。

为了说明块结构和最近嵌套规则如何影响符号表，考虑程序清单 6-3 的C代码片段。在整段代码中，有5个块。首先是整个代码的块，它包含整型变量 `i` 和 `j` 以及函数 `f` 的说明。其次，是 `f` 自身的说明，它包含参数 `size` 的说明。再次，是 `f` 函数体的复合语句，它包含字符变量 `i` 和 `temp` 的说明 (函数说明和相关的函数体也可看成表示一个块)。第四，是包含说明 `double` 的复合语句。最后，是包含说明 `char*j` 的复合语句。在函数 `f` 内部，符号表中有变量 `size` 和 `temp` 的单独说明，所有这些名字的使用都参考这些说明。对于名字 `i` 的情况，在 `f` 的复合语句

内部*i*有一个的char局部说明，根据最近嵌套规则，这个说明代替了围绕代码文件块的*i*的非局部int说明。(非局部的int *i*被称作在*f*内部有一个作用域空洞(scope hole)。)类似地，*f*中两个后来的复合语句中的*j*的说明取代它们各自块内的非局部的int说明。在每种情况中，当局部说明的块存在时，*i*和*j*的原始说明被覆盖。

程序清单6-3 说明嵌套作用域的C代码片段

```
int i,j;

int f (int size)
{ char i, temp;
  ...
  { double j;
    ...
  }
  ...
  { char * j;
    ...
  }
}
```

在许多语言中，像Pascal和Ada (但没有C)，过程和函数都可以嵌套。这对这些语言的运行环境造成了一个复杂的因素(在下一章研究)，但对嵌套作用域没有造成特别的复杂性。例如，程序清单6-4的Pascal代码包含了嵌套过程*g*和*h*，但和程序清单6-3中的C代码有本质上相同的符号表结构(当然，除了增加的名字*g*和*h*)。

程序清单6-4 说明嵌套作用域的Pascal代码片段

```
program Ex;
var i,j: integer;

function f ( size: integer ) : integer;
var i, temp: char;

  procedure g;
  var j: real;
  begin
    ...
  end;

  procedure h;
  var j: ^char;
  begin
    ...
  end;

begin (* f *)
  ...
end;

begin ( * main program * )
  ...
end.
```

为了实现嵌套作用域和最近嵌套规则，符号表插入操作不必改写前面的说明，但必须临时隐藏它们，这样查找操作只能找到名字最近插入的说明。类似地，删除操作不应删除与这个名字相应的所有说明，只需删除最近的一个，而显示前面任何的说明。然后符号表构造可以继续：执行插入操作使所有说明的名字进入每个块，执行相应的删除操作使相同的名字从块中退出。换句话说，符号表在处理嵌套作用域期间的行为类似于堆栈的方式。

为了说明这个结构如何能用实际的方法进行维护，考虑早先描述的符号表的杂凑表实现。为简单起见，假定与图6-11类似，进行程序清单6-3中过程f的主体说明之后，符号表如图6-12a所示。在处理f的主体的第2个复合语句期间(包含说明char*j)，符号表如图6-12b所示。最后，函数f的块退出之后，符号表如图6-12c所示。注意，对每个名字，每个“桶”中链接表的行为就像名字不同说明的堆栈。

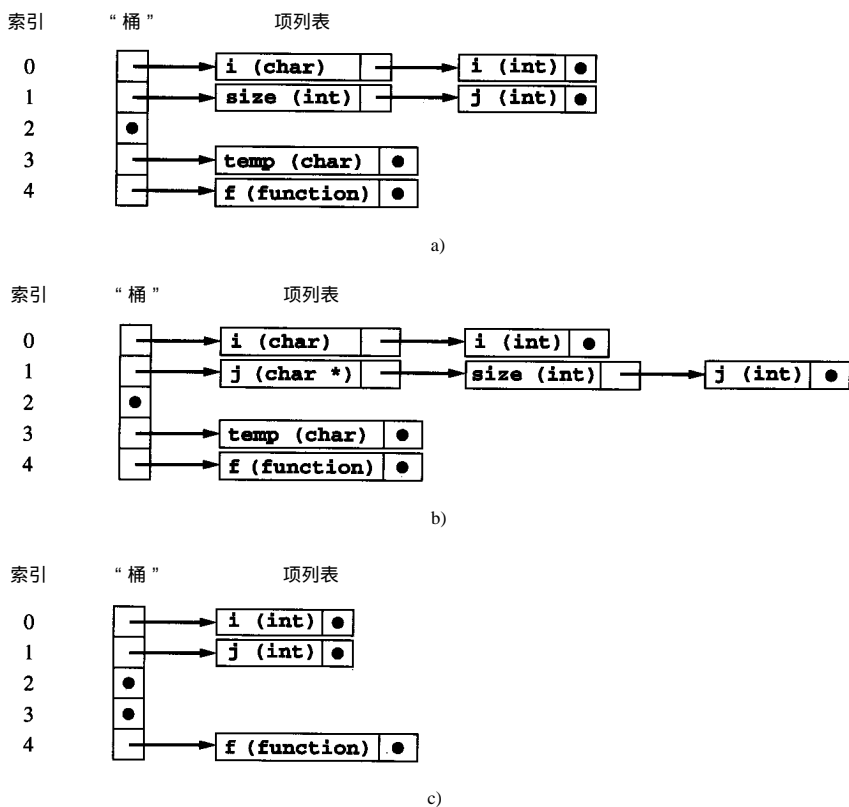


图6-12 符号表内容程序清单6-3

- a) 处理f的主体说明之后 b) 处理f的主体内第2层嵌套的复合语句的说明之后
c) 退出f的主体之后(删除其说明)

有一系列可能的方法可以实现嵌套的作用域。一种解决办法是为每个作用域建立一个新的符号表，再从内到外把它们链接在一起，这样如果查找操作在当前表中没有找到名字，就自动用附上的表继续搜索。离开作用域简单多了，不需要使用删除操作对说明再处理。相反，对应于作用域的整个符号表能在一步中释放。这个结构对应于图6-12b的一个例子在图6-13给出。在那个图中有3张表，每个作用域一个，从最内层向最外层链接。离开一个作用域只要求重设访问指针(在左边指示)指向最近的外层的作用域。

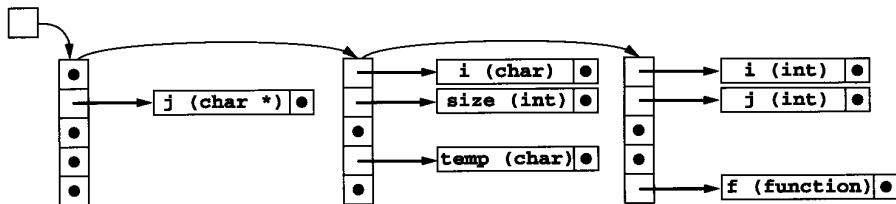


图6-13 对应于图6-12b的符号表结构，每个作用域使用独立的表

在符号表的构造期间还需要一些另外的处理和属性计算，这依赖于具体的语言和编译器操作的细节。一个例子是在Ada中要求非终结符名字在作用域空洞中仍然是可视的，通过使用类似于记录字段选择的符号可以引用它，使用与说明非终结符名字的作用域相关的名字。例如，在程序清单6-4的Pascal代码中，函数`f`内部的全局整型变量`i`，在Ada中作为变量`Ex.i`仍然是可视的(使用程序名标识全局作用域)。因此，会有这样的感觉，在构造符号表时，用名字标识每个作用域，通过累加嵌套作用域名为作用域中说明的每个名字加上前缀。因此，在程序清单6-4中名字`j`的所有出现都区分为`Ex.j`、`Ex.f.g.j`和`Ex.f.h.j`。另外或另一方面，每个作用域需要分配一个嵌套层(nesting level)或嵌套深度(nesting depth)，在每个符号表入口中记录每个名字的嵌套层。因此，在程序清单6-4中，程序的全局变量嵌套层次是0，`f`的说明(其参数和局部变量)的层次是1，`g`和`h`的说明的层次是2。

与刚描述的Ada中作用域选择特性类似的一种机制是C++中作用域限定操作符(scope resolution operator)::。这个操作符允许类说明的作用域从说明的外部进行访问。这可以用来在类说明的外面完成成员函数的说明：

```
class A
{ ... int f();...}; //是一个成员函数

A::f() //这是A 中f 的说明
{ ... }
```

类、过程(Ada中)以及记录结构都可以看作表示名字的作用域，把局部说明的集合作为一个属性。在这些作用域能被外部引用的情况下，为每个作用域建立一个单独的符号表是有好处的(就像在图6-13)。

至此我们随着程序的文本结构讨论了标准的作用域规则。有时称其为词法作用域(lexical scope)或静态作用域(static scope)(因为符号表静态地建立)。另一种作用域规则用于一些面向动态的语言(LISP、SNOBOL较早的版本，以及一些数据库查询语言)称作动态作用域(dynamic scope)。这个规则要求嵌套作用域的申请随着执行路径进行，而不是程序原来的安排。在程序清单6-5的C代码是一个简单的例子，说明两个规则的不同点。使用C语言标准的作用域规则这段代码打印出1，因为文件层变量`i`的作用域扩展到了过程`f`。如果使用动态作用域，程序就打印出2，因为`f`在中`main`调用，`main`包含了的说明(值为2)，如果使用顺序调用处理非终结符引用，它就扩展到`f`。动态作用域要求在执行时通过执行插入和删除操作建立，因为作用域也在运行时进入或退出。因此，使用动态作用域要求符号表变成环境的一部分，由编译器产生代码来维护它，而不是由编译器直接(静态地)建立符号表。动态作用域也破坏了程序的可读性，因为不模拟程序的执行就不能解决非局部的引用。最后因为变量的数据类型必须用符号表来维护，所以动态作用域与静态类型检查不兼容(注意在程序清单6-5的代码中，如果`main`内的`i`说明成`double`所出现的问题)。因此，在现代语言中动态作用域很少使用，我们不再进一步讨论。

程序清单 6-5 说明静态和动态作用域不同之处的代码

```
#include <stdio.h>

int i = 1;

void f ( void )
{ printf ( " %d\n ", i ) ; }

void main ( void )
{ int i = 2;
  f ( );
  return 0;
}
```

最后，我们要注意，在图 6-12 中的删除操作从符号表中完全消除说明。事实上有必要在表中保留说明(或至少不剥夺它们的存储区)，因为编译器的其他部分在后面可能需要引用。如果它们必须保存在符号表中，那么删除操作只需要仅仅把它们标记成不活动的，而查找操作在搜索符号表时跳过这些标记的说明。

6.3.4 同层说明的相互作用

关于作用域更深一步的问题是在同一嵌套层中(即相连到一块)说明的相互作用。对不同的说明和要翻译的语言，这些变化很大。许多语言中(C、Pascal、Ada)一个典型的要求是在同一层中说明不能使用相同的名字。因此，在C中，下面连续的说明将引起编译错误：

```
typedef int i;
int i;
```

为检查这个要求，在每次插入前编译器必须执行一次查找，通过某种机制(如嵌套层)确定在同一层中任何已存在的说明是否有相同的名字。

更困难的是在相同层的序列中名字相互之间有多少可用的信息。例如，考虑下面的C代码片段

```
int i = 1;

void f ( void )
{ int i = 2, j = i+1;
  ...
}
```

这里有一个问题：`f`内部`j`的值是初始化成2还是3，即使用的是`i`的局部说明还是非局部说明。根据最近嵌套规则，应该是使用最近的说明——局部说明。事实上这是C的方法。但是这预示在处理时每个说明加进符号表，称作顺序说明(sequential declaration)。可以替代所有要“同时”处理的说明，在说明部分的最后立即加进符号表。然后说明中任意表达式的名字将引用前面的说明，不再处理新的说明。这样的说明结构称作并列说明(collateral declaration)，一些函数式语言，像ML和Scheme，有这样的说明结构。这样的说明规则要求说明不立即加进存在的符号表中，而累加进一个新的表中(或临时结构)，在处理完所有的说明之后再加进现存的表中。

最后，是递归说明(recursive declaration)结构的情况，说明可以引用其自身或相互引用。这对于过程/函数说明特别必要，相互递归函数组是公共的(举例来说，在递归下降分析程序中)。在最

简单的形式中，递归函数调用其自身，如在下面的C代码中的函数计算两个整数的最大公因子：

```
int gcd ( int n, int m )
{ if ( m == 0 ) return n;
  else return gcd ( m, n % m );
}
```

要能正确编译它，编译器必须在处理函数体之前把函数名 `gcd` 加进符号表。否则，当递归调用遇到 `gcd` 时就找不到这个名字(或含义不正确)。在更复杂的情况中，有一组互相递归调用的函数，如下面的C代码片段

```
void f ( void )
{... g ( ) ... }

void g ( void )
{... f ( ) ... }
```

在处理函数体之前只是把每个函数加进符号表是不够的。说明上面的 C 代码编译时在 `f` 内部调用 `g` 甚至会产生错误。在 C 语言中对这个问题的解决是在 `f` 的说明之前为 `g` 加上一个称作函数原型(function prototype)的说明：

```
void g ( void ); /*函数原型说明 */

void f ( void )
{ ... g ( ) ... }

void g ( void )
{... f ( ) ... }
```

这样的说明可以看作是作用域修正(scope modifier)，扩展名字 `g` 的作用域使其包含 `f`。因此，当到达 `g` 的(第1个)原型说明时(与它自己的位置作用域属性一起)，编译器把 `g` 加进符号表。当然在到达 `g` 的主说明(或定义)之前 `g` 的函数体一直是不存在的；而且，`g` 的所有原型必须进行类型检查以确保在结构上的统一。

相互递归问题还有不同的解决办法。例如，在 Pascal 中提供了向前(forward)说明作为过程/函数作用域的扩展。在 Moudle-2 中，过程和函数(还有变量)的作用域规则扩展它们的作用域包含整个说明块，这样就自然进行相互递归调用而不必使用另外的语言机制。这就要求一个处理步骤，所有的过程和函数在处理其主体的任何部分之前加进符号表。类似的相互递归调用说明也可用于一些其他的语言。

6.3.5 使用符号表的属性文法的一个扩充例子

现在考虑一个例子，来演示我们已描述过的说明的一些特性，并研究一个属性文法使这些特性在符号表的行为清晰呈现。这个例子中使用的文法是浓缩了简单算术表达式文法，包括了对说明的扩充：

$$\begin{aligned}
 S & \quad exp \\
 exp & \quad (exp) \mid exp + exp \mid id \mid num \mid let \ dec-list \ in \ exp \\
 dec-list & \quad dec-list, decl \mid decl \\
 decl & \quad id = exp
 \end{aligned}$$

因为属性文法包含层次属性，因此就需要在语法树的根节点进行初始化，这个文法包括一个顶层的开始符号 `S`。这个文法只有一个操作符(加，记号为 `+`)，所以非常简单。它也是二义性的，我们假定分析器已经构造了语法树，或者已经处理了二义性(我们把等价的无二义性文法留作练习)。不过可以包含括号，这样如果愿意，就可以用这个文法写出无二义性的表达式。就像

在前一个例子中使用的类似的文法，我们假定 *num* 和 *id* 是记号，其结构由扫描器确定（假定 *num* 是一个数字序列，*id* 是字符序列）。

包含说明的文法增加的是 *let* 表达式 (*let expression*)：

$$exp \quad let \ dec-list \ in \ exp$$

在 *let* 表达式中，说明由通过逗号分开的、形如 *id* = *exp* 的说明序列组成，一个例子是

$$let \ x = 2+1, \ y = 3+4 \ in \ x + y$$

非正式地，*let* 表达式的语义如下。*let* 记号后面的说明是建立表达式的名字，当这些名字出现在记号 *in* 后面的 *exp* 中时代表它们表示的表达式的值 (*exp* 是 *let* 表达式的主体 (body))。*let* 表达式的值是主体的值，其计算方法是用相应 *exp* 的值代替说明中的每个名字，并根据语义规则计算主体的值。例如，在前一个例子中，*x* 代表值 3 (2+1 的值)，*y* 代表值 7 (3+4 的值)。因此，表达式自身的值是 10 (等于 *x+y* 的值，*x* 的值是 3，*y* 的值是 7)。

从刚给出的语义中，我们看到说明在 *let* 表达式中表示一种常量说明 (或约束)，而 *let* 表达式表示这个语言的块。为完成这些表达式语义的非正式讨论，需要描述 *let* 表达式中的作用域规则和说明的相互作用。注意，这个文法允许仲裁 *let* 表达式相互之间的嵌套，例如在表达式

```
let x = 2, y = 3 in
  ( let x = x+1, y = ( let z=3 in x+y+z )
    in ( x+y )
  )
```

中，我们为 *let* 表达式的说明建立下列作用域规则。首先，在相同的 *let* 表达式中不能说明相同的名字，因此，形如

$$let \ x=2, \ x=3 \ in \ x+1$$

的表达式是非法的，将导致出错。其次，如果任意一个名字没有在某一个外围 *let* 表达式中说明，也将导致错误。因此，表达式

$$let \ x=2 \ in \ x+y$$

是错误的。再次，在 *let* 表达式中每个说明的作用域按照块结构的最近嵌套规则扩充到 *let* 的主体之外。因此，表达式

$$let \ x=2 \ in \ (\ let \ x=3 \ in \ x)$$

的值是 3 而不是 2 (因为在内层的 *let* 表达式中的 *x* 引用说明 *x*=3，而不是说明 *x*=2)。

最后。在顺序的相同 *let* 表达式说明的列表中，我们说明了说明的相互作用。即每个说明使用前一个说明来处理其表达式中的名字。因此，在表达式

$$let \ x=2, y=x+1 \ in \ (\ let \ x=x+y, \ y=x+y \ in \ y)$$

中，第一个 *y* 的值是 3 (使用前一个 *x* 的说明)，第 2 个 *x* 的值是 5 (使用括起来的 *let* 的说明)，第 2 个 *y* 的值是 8 (使用括起来的 *y* 的说明和刚说明 *x* 的值)。因此，整个表达式的值是 8。类似地我们请读者计算前面三重嵌套的 *let* 表达式的值。

现在我们要开发属性等式，使用符号表记录 *let* 表达式中的说明并表示刚描述的作用域规则和相互作用。为简单起见，我们仅用符号表确定表达式是否是错的。我们不写出计算表达式的值的等式，而把它留作练习。作为替代，我们计算布尔值的合成属性 *err*，根据前面说明的规则，如果表达式是错的其值为 *true*，如果表达式正确其值为 *false*。为实现这一点，需要两个继承属性，*syntab* 表示符号表，*nestlevel* 确定两个说明是否在相同的 *let* 块内。*nestlevel* 的值是一个非负整数，表示块的当前嵌套层。在最外层它的值初始化为 0。

symtab 属性需要一般的符号表操作。因为要写属性等式，在某种程度上表达符号表操作无须间接的结果，编写插入操作像参数一样操作符号表，返回一个新的符号表加入新的信息，而原始的符号表没有改变。因此，*insert (s, n, l)* 返回一个新的符号表，包含来自符号表 *s* 的所有信息，另外把名字 *n* 与嵌套层 *l* 相联系，而不改变 *s* (因为我们只确定正确性，不必要联系 *n* 的值，只是一个嵌套层)。因为这个说明保证能恢复原始的符号表 *s*，就无须一个明确的删除操作。最后，为了测试必须满足正确性的两个准则 (出现在表达式中的所有名字必须在前面说明，而且在同层中不出现重复说明)，必须能测试符号表中一个名字的出现，也能取出与出现的名字相关的嵌套层。用两个操作实现这一点，*isin (s, n)* 返回一个布尔值，决定 *n* 是否在符号表 *s* 中，*lookup (s, n)* 返回一个整数值，如果存在它给出 *n* 最近说明的嵌套层，或者如果 *n* 不在符号表 *s* 中 (这将允许使用 *lookup* 表示等式，而不首先执行 *isin* 操作) 其值为 -1。最后，必须说明初始符号表中没有入口，我们把这写作 *emptytable*。

对符号表使用这些操作和转换，现在写出表达式的 *symtab*、*nestlevel* 和 *err* 3 个属性的属性等式。完整的属性文法在表 6-9 中。

表 6-9 带有 let 块的表达式的属性文法

文法规则	语义规则
$S \rightarrow exp$	$exp.symtab = emptytable$ $exp.nestlevel = 0$ $S.err = exp.err$
$exp \rightarrow exp_1 + exp_2$	$exp_2.symtab = exp_1.symtab$ $exp_3.symtab = exp_1.symtab$ $exp_2.nestlevel = exp_1.nestlevel$ $exp_3.nestlevel = exp_1.nestlevel$ $exp_1.err = exp_2.err \text{ or } exp_3.err$
$exp \rightarrow (exp_2)$	$exp_2.symtab = exp_1.symtab$ $exp_2.nestlevel = exp_1.nestlevel$ $exp_1.err = exp_2.err$
$exp \rightarrow id$	$exp.err = \text{not } isin (exp.symtab, id.name)$
$exp \rightarrow num$	$exp.err = \text{false}$
$exp \rightarrow \text{let } dec\text{-list in } exp_2$	$dec\text{-list.intab} = exp_1.symtab$ $dec\text{-list.nestlevel} = exp_1.nestlevel + 1$ $exp_2.symtab = dec\text{-list.outtab}$ $exp_2.nestlevel = dec\text{-list.nestlevel}$ $exp_1.err = (dec\text{-list.outtab} = errtab) \text{ or } exp_2.err$
$dec\text{-list}_1 \rightarrow dec\text{-list}_2, decl$	$dec\text{-list}_2.intab = dec\text{-list}_1.intab$ $dec\text{-list}_2.nestlevel = dec\text{-list}_1.nestlevel$ $decl.intab = dec\text{-list}_2.outtab$ $decl.nestlevel = dec\text{-list}_2.nestlevel$ $dec\text{-list}_1.outtab = decl.outtab$
$dec\text{-list} \rightarrow decl$	$decl.intab = dec\text{-list.intab}$ $decl.nestlevel = dec\text{-list.nestlevel}$ $dec\text{-list.outtab} = decl.outtab$

(续)

文法规则

语义规则

decl id = exp

```

exp.symtab = decl.intab
exp.nestlevel = decl.nestlevel
decl.outtab =
    if (decl.intab = errtab) or exp.err
    then errtab
    else if lookup(decl.intab, id.name) =
        decl.nestlevel
    then errtab
    else insert(decl.intab, id.name, decl.nestlevel)

```

在最顶层，分配了两个继承属性值而留下了合成属性的值。因此，文法规则 $S \rightarrow exp$ 有3个相关的属性等式

```

exp.symtab = emptytable
exp.nestlevel = 0
S.err = exp.err

```

文法规则 $exp \rightarrow (exp)$ 也有类似的规则。

对于规则 $exp_1 \rightarrow exp_2 + exp_3$ (像通常一样，在写属性等式时对非终结符编号)，下面的规则表示右边的表达式从左边的表达式继承了属性 *symtab* 和 *nestlevel*，并且如果右边的表达式包含了至少一个错误，左边的表达式也就包含一个错误：

```

exp2.symtab = exp1.symtab
exp3.symtab = exp1.symtab
exp2.nestlevel = exp1.nestlevel
exp3.nestlevel = exp1.nestlevel
exp1.err = exp2.err or exp3.err

```

仅当在当前的符号表中不能找到名字 *id* 时，规则 $exp \rightarrow id$ 产生一个错误(我们把标识符的名字写成 *id.name*，并假定它由扫描器或分析程序计算)，这样相关的属性等式是

```
exp.err = not isin (exp.symtab, id.name)
```

另一方面，规则 $exp \rightarrow num$ 永远不会产生错误，因此属性等式是

```
exp.err = false
```

现在开始讨论 *let* 表达式，使用文法规则

```
exp1 let dec-list in exp2
```

和相关的说明规则。在 *let* 表达式的规则中，*dec-list* 由一系列必须加进当前符号表的说明组成。通过用 *dec-list* 联系两个独立的符号表来表示这一点：从 exp_1 继承的输入表 *intab*，以及输出表 *outtab*，它包含必须传递到 exp_2 的新的(和旧的)说明。因为新的说明可能包含错误(如重复说明相同的名字)，必须考虑从 *dec-list* 传递的一个特别的 *errtab* 符号表。最后，当 *dec-list* 包含一个错误(这种情况 *outtab* 出错)或者 *let* 表达式的主体 exp_2 包含一个错误，*let* 表达式也出错。属性等式是


```

dec-list.intab = exp1.syntab
dec-list.nestlevel = exp1.nestlevel + 1
exp2.syntab = dec-list.outtab
exp2.nestlevel = dec-list.nestlevel
exp1.err = (dec-list.outtab = errtab) or exp2.err

```

注意当进入 *let* 块时嵌套层也增加1。

还剩下为说明列表和单个的说明开发等式。根据说明的顺序性规则，说明列表必须在处理列表过程中累积。因此，给定规则

$$dec-list_1 \quad dec-list_2, decl$$

像 *intab* 传递到 *decl* 一样，*outtab* 从 *dec-list₂* 传递，因此给定 *decl* 访问说明先于它在列表中。对于单个说明的情况 (*dec-list decl*)，执行了标准的继承和移位。完整的等式在表 6-9 中。

最后，讨论单个说明的情况

$$decl \quad id = exp$$

在这个情况中继承属性 *decl.intab* 立即传递到 *exp* (因为在这个语言中说明是非递归的，因此 *exp* 必须找到这个 *id* 名字的前一个说明，而不是当前的那个)。然后，如果没有错误，*id.name* 用当前嵌套层插入到表中，并作为说明继承 *outtab* 传递回去。在 3 种情况会出现错误。首先，在前面的说明中已经有了错误，这种情况是：*decl.intab* 是 *errtab*，*errtab* 必须作为 *outtab* 传递。其次，错误可能在 *exp* 中出现：如果是这样，则由 *exp.err* 指出，*outtab* 也会引起变成 *errtab*。再次，说明可能是一个名字在同一嵌套层次中的重复说明。这必须通过执行一次 *lookup* 来检查，这个错误也必须把 *outtab* 强制变成 *errtab* 来报告 (注意，如果 *lookup* 没有找到 *id.name*，则返回 -1，在当前的嵌套层没有匹配，因此没有错误产生)。*decl.outtab* 完整的等式在表 6-9 中给出。

这就完成了属性等式的讨论。

6.4 数据类型和类型检查

编译器的主要任务之一是数据类型信息的计算和维护 (类型推论 (type inference)) 以及使用这些信息确保程序的每一部分在语言的类型规则作用下有意义 (类型检查 (type checking))。通常地，这两个任务密切相关并一起执行，但只提及类型检查。数据类型信息可以是静态的或动态的或是两者的混合。在大多数 LISP 语系的语言中，类型信息完全是动态的。在这样的语言中，编译器必须在执行时产生代码完成类型推论和类型检查。在大多数传统语言中，如 Pascal、C 和 Ada，类型信息主要是静态的，主要在程序执行之前进行正确性检查。静态类型信息也用来确定每个变量分配所需要的存储器大小以及存储器的访问方式，这可以用来简化运行环境 (将在下一章讨论这一点)。这一节将只关心静态数据类型。

数据类型信息可以用几种不同的形式出现在程序中。在理论上，数据类型 (data type) 是值的集合，更精确一点，是那些值上某几种操作的值的集合。例如，数据类型 *integer* 在一个编程语言中指的是数学整数的子集，以及算术操作，如 + 和 *，由语言说明提供。在编译器构造的实际领域，这些集合通常用类型表达式 (type expression) 描述，有一个类型名，如 *integer*，或结构表达式，如 *array [1..10] of real*，其操作通常假定或隐含。类型表达式在一个程序中可能出现几次。这些表达式包括变量说明，如

```
var x: array [1..10] of real;
```


它把类型与一个变量名和类型说明相关，又如

```
type RealArray = array [1..10] of real;
```

它说明一个新的类型名，用于以后的类型或变量说明。这样的类型信息是明确的。类型信息也可能是隐含的，如Pascal中常量说明的例子

```
const greeting = "Hello!";
```

这里根据Pascal的规则，`greeting`隐含说明为`array [1..6] of char`类型。

包含在说明中显式或隐含定义的类型信息保持在符号表中，当引用相关的名字时，由类型检查器取出，新的类型则从这些类型中推断出来，并和语法树中相应的节点关联。例如，在表达式

```
a [ i ]
```

中名字`a`和`i`的数据类型从符号表中取出，如果`a`的类型是`array [1..10] of real`，而`i`的类型是`integer`，那么子表达式`a[i]`的类型为`real`，并被判断为正确的(`i`的值是否在1和10之间的问题是范围检查(range checking)问题，不能像通常一样静态确定)。

编译器表示数据类型的方式、符号表维护类型信息的方式以及类型检查器推断类型使用的规则等，都依赖于语言中可用的类型表达式的种类和语言中管理这些类型表达式使用的类型规则。

6.4.1 类型表达式和类型构造器

编程语言通常包含一些内嵌的类型，如`int`和`double`。这些预说明(predefined)类型或者对应于由各种机器体系结构内部提供的数字数据类型，其操作作为机器指令已经存在，或者是像`boolean`或`char`一样属性很容易实现的基本类型。这些数据类型是简单类型(simple type)，其值呈现为无明确的内部结构。对于整数一种典型的表示是2字节，或4字节作为2字节的补充形式。实数或浮点数的典型表示，是4或8字节数，带一个符号位，一个指数字段和一个分数(或尾数)字段。字符的典型表示是一字节的ASCII代码，对布尔值是一个字节，只使用最低的一位(1=true, 0=false)。有时语言也在如何实现这些预说明类型上加强限制。例如，C语言标准要求`double`浮点数类型至少有10位十进制数字的精度。

在C语言中一种有趣的预说明类型是`void`类型。这个类型没有值，因此表示空的集合。它用于表示没有返回值的函数(即过程)，也可表示一个指针指向未知类型。

在一些语言中，可以说明一些新的数据类型。典型的例子是子界类型(subrange type)和枚举类型(enumerated type)。例如，在Pascal语言中由0~9组成的整数的子界类型可以说明为

```
type Digit = 0..9;
```

在C语言中由名字为`red`、`green`和`blue`组成的枚举类型说明为

```
typedef enum { red, green, blue } Color;
```

子界和枚举都可以作为整数实现，或者使用较小的足以表示所有值的存储器。

给定一个预说明类型的集合，使用类型构造器(type constructor)，如数组(array)、记录(record)和结构(struct)，可以创建新的类型。这些构造可以看作是函数，把存在的类型作为参数，而用依赖于构造的一个结构返回新的类型。这些类型通常称作结构类型(structured type)。在这些类型的分类中，了解类型表示的值的集合的特性是重要的。通常，在类型构造的参数值的基本集上，它密切对应于一组操作。我们通过列出一些公共的构造并把它们与集合操作比较来说明这一点。

1) 数组 数组类型构造有两个类型参数, 一个是索引类型(index type), 另一个是元素类型(component type), 并产生一个新的数组类型。在类Pascal语言中我们写作

array [索引类型] of 元素类型

例如, Pascal类型表达式

```
array [Color] of Char;
```

创建一个数组类型, 其索引类型是**Color**, 元素类型是**Char**。通常对于索引类型有一些限制。例如, 在Pascal语言中索引类型限制为所谓的序数类型(ordinal types): 这些类型的每个值都有一个直接前驱和直接后继。这样的类型包括整数和字符的子界类型以及枚举类型。对照地在C语言中, 整数作用域只允许从0开始, 并只能指定大小来代替索引类型。事实上在C语言中没有关键字对应于**array**, 只是仅仅加上在括号表示作用域的后缀来说明数组类型。因此对前面Pascal的类型表达式在C中没有直接的等式, 但C的类型说明

```
typedef char Ar [3];
```

说明了**Ar**类型, 是与前面的类型等价的类型结构(假定**Color**有3个值)。

数组表示的是元素类型的值的序列, 并由索引类型的值进行索引。也就是说, 如果索引类型有值**I**的集合, 元素类型有值**C**的集合, 那么对应于类型**array [索引类型] of 元素类型**值的集合是通过**I**的元素索引的**C**的元素有限序列的集合, 或者在数学项中, 函数**I → C**的集合。数组类型的值的相关操作由单个下标的操作组成, 它可以用于给元素赋值或从元素中取出值:

```
x := a[red] 或 a[blue] := y
```

数组一般根据索引从小到大分配连续的存储空间, 允许在执行期间使用自动的偏移量计算。所需存储空间的大小是 $n * size$, 这里 n 是在索引类型中值的数目, $size$ 是元素类型的一个值所需存储器的大小。因此, 如果每个整数占据4个字节, 类型**array [0..9] of integer**的一个变量需要40字节的存储空间。

数组说明中的一种复杂情况是多维数组(multidimensioned arrays)。这通常可以通过重复应用数组类型构造来说明, 如

```
array [0..9] of array [Color] of integer
```

或者进行简化, 把索引集合列在一起:

```
array [0..9, Color] of integer
```

在第1种情况中出现的下标如**a[1][red]**, 第2种情况下写成**a[1, red]**。多重下标的一个问题是在存储器中值的序列可以用不同的方式, 索引过程是: 首先是第一个索引, 然后是第2个索引, 或者相反。用第一种索引方式的结果是在存储器中值的顺序是**a[0, red], a[1, red], a[2, red], ..., a[9, red], a[0, blue], a[1, blue], ..., a[9, blue]**等等(这称作列前提形式(column-major form)), 用第2种索引方式的结果是在存储器中值的顺序是**a[0, red], a[0, blue], a[0, green], a[1, red], a[1, blue], a[1, green], ..., a[9, red], a[9, blue], a[9, green]**等等(这称作行前提形式(row-major form))。如果多维数组重复说明的版本和简化版本是等价的(也就是说**a[0, red] = a[0][red]**), 那么必须使用行前提形式, 因为可以分开加上不同的索引: **a[0]**的类型必须是**array [Color] of integer**而且必须指向一个连续的存储区。FORTRAN语言, 没有多维数组的局部索引, 传统上使用列前提形式实现。

有时候, 语言允许使用没有说明索引作用域的数组。这样的开放索引数组(open-indexed array)对函数的数组参数说明特别有用, 这样函数可以处理不同大小的数组。例如, C语言说明

```
void sort ( int a[], int first, int last )
```

可以用来说明一个分类程序，用于任意大小的数组 a (当然，在调用时必须使用一些方法来确定实际的大小。在这个例子中，使用了其他的参数)。

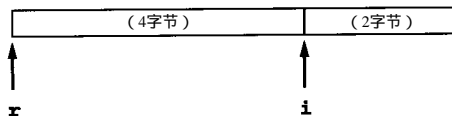
2) 记录 记录(record)或结构(structure)类型构造器接受一个名字列表和相关的类型并构造一个新的类型。如在C语言中

```
struct
{ double r;
  int i;
}
```

记录与数组不同，不同类型的元素可以组合起来(在数组中所有的元素都有相同的类型)，使用名字(而不是索引)访问不同的元素。记录类型的值大致对应于其元素类型值的笛卡儿积，使用名字而不是位置访问元素。例如，前面给定的记录大致对应于笛卡儿积 $R \times I$ ，这里 R 是相应的 `double` 类型的数据集合， I 是相应的 `int` 类型的数据集合。更准确地，给定的记录对应于笛卡儿积 $(r \times R) \times (i \times I)$ ，这里名字 r 和 i 识别各个元素。这些名字通常使用圆点符号(dot notation)选择表达式中相应的元素。因此，如果 x 是给定记录类型的一个变量，那么 $x.r$ 表示第1个元素， $x.i$ 表示第2个元素。

一些语言具有纯的笛卡儿积类型构造器。这样的一种语言是 ML，这里 `int*real` 是整数和实数笛卡儿积的符号。类型 `int*real` 的值写成两个一组，如 $(2, 3.14)$ ，元素通过投影函数 `fst` 和 `snd` (表示第1和第2)访问：`fst(2, 3.14) = 2` 和 `snd(2, 3.14) = 3.14`

记录或笛卡儿积的标准实现方法是顺序地分配存储器，为每个元素类型分配一块存储器。因此，如果一个实数需要4个字节，一个整数需要两个字节，那么前面给定的记录结构需要6个字节的存储器，分配如下：

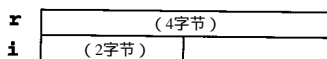


3) 联合 联合类型对应于联合操作集合。在C语言中它可以直接通过 `union` 说明使用，例如说明

```
union
{ double r;
  int i;
}
```

说明了实数和整数的联合类型。严格地讲，这是一个脱节的联合(disjoint union)，因为每个值可以看成是实数或整数，但不能同时是两者。通过访问值的元素名字可以解释的更清楚一些：如果 x 是给定的联合类型的一个变量，那么 $x.r$ 表示 x 是实数时的值，而 $x.i$ 表示 x 是整数时的值。在数学上联合的说明写成 $(r \times R) \cup (i \times I)$ 。

联合的标准实现方法是为每个元素并行地分配存储器，这样每个元素类型的存储器与所有其他的类型相重叠。因此，如果一个实数需要4个字节，一个整数需要两个字节，那么前面给定的联合结构仅需要4个字节的存储器(其元素所需的最大的存储器)，存储器分配如下：



事实上, 这样的实现需要联合被解释成脱节的联合, 因为整数的表示不会适合相应的实数的表示。实际上, 编程者区分这些值没有什么意义, 这样的联合在数据解释中会导致错误, 也提供了一种方法绕过类型检查器。例如, 在C语言中如果 x 是给定的联合类型的一个变量, 那么

```
x.r = 20;
printf ( "%d", x.i );
```

将引起一个编译错误, 不会打印出值2, 而是一个垃圾值。

在联合类型中的这种不安全性已由许多不同的语言设计者处理。例如在Pascal中, 联合类型使用一种所谓的可变记录 (variant record) 说明, 其中序数类型的值记录在一个判别式 (discriminant) 元素中并用来区分想要的值。因此, 在Pascal中前面的联合类型可以写成

```
record case isReal: boolean of
    true: (r: real);
    false: (i: integer);
end;
```

现在这个类型的变量 x 有3个元素: $x.isReal$ (布尔值)、 $x.r$ 和 $x.i$, 并且 $isReal$ 字段在存储器中分配一个独立的非覆盖的空间。当赋一个实数值时, 如 $x.r := 2.0$, 同时也要赋值 $x.isReal := true$ 实际上, 这个机构相对而言是没有用的 (至少在类型检查时编译器的使用), 因为判别式的赋值和值的辨别可以分开。当然, Pascal允许通过在说明中删除其名字说明判别式元素 (但不使用其值区分 case), 如在

```
record case bollean of
    true:(r: real);
    false:(i: integer);
end;
```

中不再为判别式分配存储空间。因此, Pascal编译器在合法性检查时几乎从不进行任何尝试使用判别式。另一方面, Ada有类似的机制, 但是强调只要一个联合元素被赋值, 判别式必须同时赋值进行合法性检查。

在像ML这样的函数式语言中采用了一种不同的方法, 联合类型的说明使用了一根竖线表示联合, 为每个元素给定一个名字来区分它们, 如:

```
IsReal of real | IsInteger of int
```

现在当使用相应的类型的值时也必须使用名字 $IsReal$ 和 $IsInteger$, 就像在 ($IsReal\ 2.0$) 或 ($IsInteger\ 2$) 中一样。名字 $IsReal$ 和 $IsInteger$ 称作值构造器 (value constructor), 因为它们“构造”了这个类型的值。因为它们参照这个类型的值时总必须使用, 不会有解释错误发生。

4) 指针 指针类型由引用另一个类型值的值组成。因此, 指针类型的值是一个存储器地址, 其中保存着其基类型的值。指针类型经常被看成是数字类型, 因为在其上可以进行算术运算, 如加上偏移量, 乘上比例因子等。然而它们不是真正的简单类型, 因为它们是应用指针类型构造器从已有的类型中构造出来的。它也没有标准的集合操作直接对应于指针类型构造器, 就像笛卡儿积对应于记录构造器一样。指针类型在类型系统中占有比较特殊的位置。

在Pascal中字符 \wedge 对应于指针类型构造器, 因此类型表达式 $\wedge integer$ 表示“整数的指针”。在C中, 等价的类型表达式是 $int*$ 。指针类型值上的标准基本操作是解除引用 (dereference) 操作。例如, 在Pascal中, \wedge 表示解除引用操作符 (和指针类型构造器), 如果 p 是类型 $\wedge integer$ 的一个变量, 那么 p^\wedge 是 p 解除引用的值, 类型为 $integer$ 。C中也有类似的规则, $*$ 解除指针变量的引用, 并写成 $*p$ 。

指针类型在描述递归类型时最有用，我们简要地进行讨论。在这里，指针类型构造器最常用于记录类型。

指针类型基于目标机器的地址的大小分配空间。通常是4字节，有时是8字节。有时机器的体系结构强制更复杂的分配方案。例如，在基于DOS的PC中，要区别近指针(段内地址，2字节)和远指针(段外地址，4字节)。

5) 函数 我们已经注意到数组可以看成从索引集到元素集的函数。许多语言(但不是Pascal或Ada)都有描述函数类型更一般的能力。例如在Modula-2中说明

```
VAR f: PROCEDURE (INTEGER): INTEGER;
```

说明变量 f 是函数(或过程)类型，带有一个整数参数，并产生一个整数结果。在数学符号中，这个集合描述成函数 $\{f: I \rightarrow I\}$ ，这里 I 是整数的集合。在ML语言中，相同的类型被写成`int->int`。C语言也有函数类型，但它们必须用有些笨拙的符号写成“指向函数的指针”。例如，刚给出的Modula-2的说明写成C语言是

```
int (*f) (int);
```

函数类型按照目标机器的地址的大小分配空间。根据语言和运行时环境的组织方式的不同，函数类型需要给代码指针分配空间(指向实现函数的代码)或给代码指针和环境指针分配空间(指向运行环境中的位置)。环境指针的作用将在下一章讨论。

6) 类 大多数面向对象的语言都有类似于记录说明的类说明，它所包含的操作说明除外，那称作方法(method)或成员函数(member function)。类说明在一个面向对象的语言中可以创建或不创建新的类型(在C++中创建)。即使这样，类说明不仅仅是类型，因为它们允许使用属于类型系统的特性，如继承和动态联编^①。这些后期的特性必须通过独立的数据结构维护，如类继承(class hierarchy)(直接非循环图)，用于实现继承性，以及虚拟方法表(virtual method table)，用于实现动态联编。下一章我们将再次讨论这些结构。

6.4.2 类型名、类型说明和递归类型

具有丰富类型构造器的语言通常也给编程者提供一个机制给类型表达式赋名。这样的类型说明(type declaration)(有时也称作类型说明(type definition))包括C语言中的`typedef`机制和Pascal语言中的类型说明。例如C语言代码

```
typedef struct
{ double r;
  int i;
} RealIntRec;
```

说明名字`RealIntRec`作为记录类型的名字，它由在其之前的`struct`类型表达式构造。在ML语言中，类似的说明是(但没有字段名)：

```
type RealIntRec = real*int;
```

C语言有附加的类型命名机制，名字可以直接用`struct`或`union`构造器关联，而无须直接使用`typedef`。例如，C代码

```
struct RealIntRec
{ double r;
```

① 在一些语言中，如C++，继承是类型系统的镜像，因为子类可以看作是子类型(类型 S 可以看作是类型 T 的子类型，如果它所有的值都可看成是 T 的值，或者，在集合术语中，如果 $S \subseteq T$)。


```
int i;
};
```

也说明了类型名 `RealIntRec`，但它在变量说明中必须使用 `struct` 构造器名：

```
struct RealIntRec x; /*说明x 是一个RealIntRec类型的变量 */
```

就像变量说明使变量名进入符号表一样，类型说明也使说明的类型名进入符号表。产生的一个问题是否是类型名也像变量名样可以重用。通常这是不允许的（作用域嵌套规则允许除外）。对这个规则 C 语言有一个小的例外，与 `struct` 或 `union` 相关的名字可以像 `typedef` 名字一样重用：

```
struct RealIntRec
{ double r;
  int i;
};
typedef struct RealIntRec RealIntRec;
/* 一个合法的说明 */
```

通过考虑 `struct` 说明引入的类型名为整个字符串 “`struct RealIntRec`” 可实现这一点，它与 `typedef` 引入的类型名 `RealIntRec` 不同。

类型名与符号表中属性相关的方法和变量说明相似。这些属性包括作用域（它在符号表结构中可以继承）和对应于类型名的类型表达式。因为类型名可以出现在类型表达式中，与前一节讨论的函数的递归说明类似，就出现了类型名递归使用的问题。在现代的编程语言中这样的递归数据类型 (recursive data type) 特别重要，包括列表、树以及其他结构。

在处理递归类型方面，语言通常分成两组。第 1 组由允许在类型说明中直接使用递归的语言组成。这样的一种语言是 ML。例如。在 ML 中，包含整数的二叉搜索树可以说明为

```
datatype intBST = Nil | Node of int*intBST*intB
```

这可以看成 `intBST` 的说明，即是 `Nil` 值和整数与 `intBST` 自身两个拷贝（一个表示左子树，一个表示右子树）笛卡儿积的联合。等价的 C 语言说明（形式稍有改变）是

```
struct intBST
{ int isNull;
  int val;
  struct intBST left, right;
};
```

然而，这个说明在 C 语言中将产生一个错误消息，是由对类型名 `intBST` 的递归使用引起的。问题是这些说明没有确定分配一个 `intBST` 类型的变量所需的存储器的大小。这一类语言，如 ML，能接受这样的说明，在执行之前不需要这样的信息，提供一种一般的存储器自动分配和释放机制。这样的存储器管理工具是运行时环境的一部分，将在下一章讨论。C 语言没有这样的机制，因此必须使这样的递归类型说明是非法的。C 是第 2 组语言的代表——在类型说明中不允许直接使用递归。

对只允许间接使用递归的语言的解决办法是通过指针。在 C 语言中 `intBST` 正确的说明是

```
struct intBST
{ int val;
  struct intBST *left, *right;
};
typedef struct intBST * intBST;
```

或

```
typedef struct intBST * intBST;
struct intBST
{ int val;
  intBST left, right;
};
```

(在C语言中,递归说明需要使用递归类型名说明的 `struct` 或 `union` 形式)。在这些说明中每个类型所需的存储空间大小直接由编译器计算,但值的空间必须由编程者通过使用 `malloc` 这样的分配过程进行手工分配。

6.4.3 类型等价

给定语言可能的类型表达式,类型检查器经常需要回答何时两个类型表达式表示相同的类型。这就是类型等价(type equivalence)问题。一种语言有许多种可能的方法说明类型等价。这一节我们简要讨论类型等价最常用的形式。在这里,这样描述类型等价,当在编译器的语义分析程序中,即当函数

```
function typeEqual ( t1, t2 : TypeExp ) : Boolean;
```

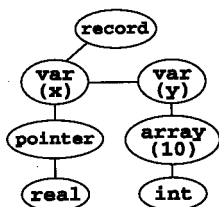
接受两个类型表达式,如果根据语言的类型等价规则它们表示相同的类型就返回 `true`,否则返回 `false`。对于不同的类型等价算法,将给出这个函数的几种不同的伪代码描述。

一种有关直接描述类型等价算法的方法是类型表达式在编译器内表示。一种简单的方法是使用语法树表示,因为这使得从说明的语法直接转换到类型的内部表示十分容易。对于这样表示的一个具体例子,考虑图 6-14 给出的类型表达式和类型说明的文法。其中有我们已讨论过的许多类型结构的简单版本。但是没有允许关联新的类型名到类型表达式的类型说明(因此不可能有递归类型,尽管出现了指针类型)。对应于图中的文法规则,要为类型表达式描述一个可能的语法树结构。

首先考虑类型表达式

```
record
  x: pointer to real;
  y: array [10] of int
end
```

这个类型表达式可以用下面的语法树表示



这里记录的子节点表示成同属列表,因为记录元素的数目是任意的。注意,表示简单类型的节点构成了树的叶子。

```
var-decls  var-decls ; var-decl | var-decl
var-decl   id : type-exp
```



```

type-exp    simple-type | structured-type
simple-type  int | bool | real | char | void
structured-type  array [num] of type-exp |
                record var-decls end |
                union var-decls end |
                pointer to type-exp |
                proc ( type-exps ) type-exp
type-exps   type-exps , type-exp | type-exp

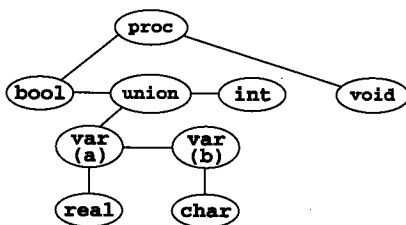
```

图6-14 类型表达式的简单文法

类似地，类型表达式

```
proc ( bool, union a:real; b:char end, int ) : void
```

可以用以下语法树表示



注意，参数类型也给定为同属列表，而结果类型（在这个例子中是 `void`）通过使它直接成为 `proc` 的一个子节点来区分。

我们描述的第一种类型等价，也是仅有的可用于缺省类型名的，是结构等价（structural equivalence）。在这个等价观点中，两个类型当且仅当它们有相同的结构时它们才相同。如果用语法树表示类型，这个规则说两个类型是等价的，当且仅当它们的语法树结构是同一的。在练习中有一个例子，说明如何检查结构等价。程序清单 6-6给出了函数 `typeEqual` 的伪代码描述，它是图 6-14 中文法给出的两个类型表达式的结构等价，使用了我们刚描述过的语法树。

我们从程序清单 6-6 的伪代码描述中注意到这个版本的结构等价意味着两个数组是不等价的，除非它们有相同的大小和元素类型，两个记录是不等价的，除非它们有相同的元素并且元素有相同的名字和顺序。在一个结构等价算法中，可能有一些不同的选择。例如，在确定等价性时数组的大小可以被忽略，也可能允许结构或联合的元素以不同的顺序出现。

当在类型说明中说明了类型表达式新的类型名时，可以说明限制性更强的类型等价。在图 6-15 中，我们修改了图 6-14 的文法以包含类型说明，同时限制变量说明和类型子表达式为简单类型和类型名。对这些说明不能再写成

```

record
    x: pointer to real;
    y: array [10] of int
end

```

而必须代替为

```
t1 = pointer to real;
```

```

t2 = array [10] of int;
t3 = record
    x: t1;
    y: t2
end

```

程序清单6-6 函数typeEqual的伪代码，测试图6-14文法类型表达式的结构等价

```

function typeEqual ( t1, t2 : TypeExp ) : Boolean;
var temp : Boolean;
    p1, p2 : TypeExp;
begin
    if t1 and t2 are of simple type then return t1 = t2
    else if t1.kind = array and t2.kind = array then
        return t1.size = t2.size and typeEqual ( t1.child1, t2.child1 )
    else if t1.kind = record and t2.kind = record
        or t1.kind = union and t2.kind = union then
        begin
            p1 := t1.child1 ;
            p2 := t2.child1 ;
            temp := true ;
            while temp and p1 ≠ nil and p2 ≠ nil do
                if p1.name ≠ p2.name then
                    temp := false
                else if not typeEqual ( p1.child1 , p2.child1 )
                then temp := false
                else begin
                    p1 := p1.sibling ;
                    p2 := p2.sibling ;
                end ;
            return temp and p1 = nil and p2 = nil ;
        end
    else if t1.kind = pointer and t2.kind = pointer then
        return typeEqual ( t1.child1 , t2.child1 )
    else if t1.kind = proc and t2.kind = proc then
    begin
        p1 := t1.child1 ;
        p2 := t2.child1 ;
        temp := true ;
        while temp and p1 ≠ nil and p2 ≠ nil do
            if not typeEqual ( p1.child1 , p2.child1 )
            then temp := false
            else begin

```

```

    p1 := p1.sibling ;
    p2 := p2.sibling ;
end;
return temp and p1 = nil and p2 = nil
    and typeEqual( t1.child2 , t2.child2 )
end
else return false;
end ; (* typeEqual *)

```

```

var-decls  var-decls ; var-decl | var-decl
var-decl   id : simple-type-exp
type-decls type-decls ; type-decl | type-decl
type-decl  id = type-exp
type-exp   simple-type-exp | structured-type
simple-type-exp  simple-type | id
simple-type    int | bool | real | char | void
structured-type  array [num] of simple-type-exp |
    record var-decls end |
    union var-decls end |
    pointer to simple-type-exp |
    proc ( type-exps ) simple-type-exp
type-exps    type-exps , simple-type-exp | simple-type-exp

```

图6-15 带类型说明的类型表达式

现在可以说明基于类型名的类型等价，这种形式的类型等价称作名等价 (name equivalence)：两个类型表达式是等价的，当且仅当它们是相同的简单类型或有相同的类型名。这是一种非常强的类型等价，因为给定类型说明

```

t1 = int;
t2 = int

```

类型 `t1` 和 `t2` 是不等价的 (因为名字不同) 对 `int` 也不等价。纯的名等价非常容易实现，因为 `typeEqual` 函数可写成以下几行：

```

function typeEqual( t1,t2 : TypeExp ) : Boolean;
var temp : Boolean ;
    p1, p2 : TypeExp ;
begin
    if t1 and t2 are of simple type then
        return t1 = t2
    else if t1 and t2 are type names then
        return t1 = t2
    else return false ;
end;

```

当然, 对应于类型名的实际的类型表达式必须进入符号表, 以允许后面为存储器分配计算存储器大小, 以及检查操作的有效性, 如指针解除引用和元素选择。

名等价中一个复杂的因素是类型表达式不同于简单类型, 或类型名在变量说明中继续被使用, 或者作为类型表达式的子表达式。在这些情况下, 类型表达式可能没有给定明确的名字, 编译器将产生一个类型表达式的中间名, 与其他任何名字都不同。例如, 给定变量说明

```
x: array [10] of int;
y: array [10] of int;
```

对应于类型表达式 `array [10] of int` 变量 `x` 和 `y` 被赋予不同的 (和唯一的) 类型名。

在出现类型名时可能保留结构等价。对这种情况, 当遇到一个名字时, 必须从符号表中取出它对应的类型表达式。这可以通过在程序清单 6-6 的代码中加进下列情况来实现,

```
else if t1 and t2 are type names then
    return typeEqual ( getTypeExp(t1), getTypeExp(t2))
```

这里 `getTypeExp` 是一个符号表操作, 返回与其参数 (必须是一个类型名) 相关的类型表达式结构。这要求每个类型名必须用表示其结构的类型表达式插入到符号表, 或者至少由类型说明产生类型名的链, 如

```
t2 = t1;
```

在符号表中最后带回类型结构。

当可能有递归类型引用时, 实现结构等价必须小心, 因为刚才描述的算法会导致无限循环。通过改变调用 `typeEqual (t1, t2)` 的方法可以避免这种情况, 这里 `t1` 和 `t2` 是类型名, 假定它们已经潜在地等价。然后如果函数曾经返回相同的调用, 在那里就可说明成功。例如, 考虑类型说明

```
t1 = record
    x: int;
    t: pointer to t2;
end;
t2 = record
    x: int;
    t: pointer to t1;
end;
```

给定调用 `typeEqual (t1, t2)`, 函数 `typeEqual` 将假定 `t1` 和 `t2` 潜在地等价。然后取出 `t1` 和 `t2` 的结构, 并且算法将成功进行直到调用 `typeEqual (t2, t1)` 分析指针说明的子孙类型。这个调用将立即返回 `true`, 因为在初始调用中已经假定了它们潜在地等价。通常, 这个算法需要进行成功地假设, 那一对类型名是相等的, 并在一个列表中累积假设。最后, 当然, 算法或者成功, 或者失败 (即它不会无限循环), 因为在任何给定的程序中只有有限的类型名。我们把 `typeEqual` 伪代码的修改细节留作练习 (见注意与参考节)。

类型等价最后一个变化是 Pascal 和 C 使用的名等价的一个弱化的版本, 称作说明等价 (declaration equivalence)。在这个方法类型中, 像

```
t2 = t1;
```

这样的说明是作为类型别名 (alias) 解释的, 而不是新的类型 (作为名等价中)。因此, 给定说明

```
t1 = int;
t2 = int
```

`t1`和`t2`对`int`等价(即它们仅是类型名`int`的别名)。在这个类型等价版本中,每个类型名等价于某个基类型名,它或者是一个预说明类型,或者是由类型构造器产生的类型表达式给定的。例如,给定说明

```
t1 = array [10] of int;
t2 = array [10] of int;
t3 = t1;
```

类型名`t1`和`t3`根据说明等价是等价的,但和`t2`都不等价。

为实现说明等价,符号表必须提供一个新的操作 `getBaseTypeName`,它取出基类型名而不是相关的类型表达式。在符号表中,一个类型名如果是预说明类型或由类型表达式给出,而不仅是另一个类型名,它就被区分为基类型名。注意,说明等价类似于名等价,在检查递归类型时解决无限循环问题,因为如果两个基类型名有相同的名字只能是说明等价。

Pascal一律使用说明等价,而C对结构和联合使用说明等价,但对指针和数组使用结构等价。

有时,一种语言将提供结构、说明或名等价,对不同的类型说明使用不同形式的等价。例如,ML语言允许使用保留关键字 `type` 把类型名说明为别名,如说明

```
type RealIntRec = real*int;
```

这把`RealIntRec`说明成笛卡儿积类型`real*int`的别名。另一方面,说明完全创建了一个新的类型,如说明

```
datatype intBST = Nil | Node of int*intBST*intBST
```

注意,`datatype`说明也必须包含值构造器名(在给定的说明中是`Nil`和`Node`)。而不像`type`说明。这使新类型的值能从已存在的类型的中区分出来。因此,给定说明

```
datatype NewRealInt = Prod of real*int;
```

值`(2.7,10)`是类型`RealIntRec`或`real*int`的,而值`Prod(2.7,10)`是类型`NewRealInt`的(而不是`real*int`)。

6.4.4 类型推论和类型检查

现在,基于类型的表示和前一节讨论的 `typeEqual` 操作,我们对一个简单语言的语义分析动作方面的类型检查器进行描述。使用的语言具有图 6-16 给定的文法,包括图 6-14 中类型表达式的一个小的子集,加上了少量的表达式和语句。我们还假定符号表的可用性包括变量名和相关的类型,插入操作,在表中插入名字和类型,及查找操作,返回名字的相关类型。在属性文法中我们将不指定这些操作本身的特性。我们将分别讨论每种语言构造的类型推断和类型检查规则。语义动作的完整列表在表 6-10 中给出。这些动作没有用纯的属性文法形式给出,并且使用符号`:=`而不是表 6-10 规则中的等号来指示。

```
program    var-decls ; stmts
var-decls  var-decls ; var-decl | var-decl
var-decl   id : type-exp
type-exp   int | bool | array #um]of type-exp
stmts      stmts ; stmt | stmt
stmt       if exp then stmt | id := exp
```

图6-16 说明类型检查的简单文法

1) 说明 说明引起标识符的类型进入符号表。因此,文法规则

$$var-decl \quad \mathbf{id} : type-exp$$

有相应的语义动作

```
insert ( id .name, type-exp.type)
```

把标识符插入到符号表并关联一个类型。在这个插入中相关的类型根据 *type-exp* 的文法规则构造。

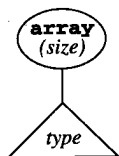
表6-10 用于图6-16 简单文法类型检查的属性文法

文法规则	语义规则
<i>var-decl</i> id : <i>type-exp</i>	<i>insert</i> (id .name, <i>type-exp.type</i>)
<i>type-exp</i> int	<i>type-exp.type</i> := <i>integer</i>
<i>type-exp</i> bool	<i>type-exp.type</i> := <i>boolean</i>
<i>type-exp</i> ₁ array [num] of <i>type-exp</i> ₂	<i>type-exp</i> ₁ .type := <i>makeTypeNode</i> (<i>array.num.size</i> , <i>type-exp</i> ₂ .type)
<i>stmt</i> if <i>exp</i> then <i>stmt</i>	if not <i>typeEqual</i> (<i>exp.type</i> , <i>boolean</i>) then <i>type-error</i> (<i>stmt</i>)
<i>stmt</i> id := <i>exp</i>	if not <i>typeEqual</i> (<i>lookup</i> (id .name), <i>exp.type</i>) then <i>type-error</i> (<i>stmt</i>)
<i>exp</i> ₁ <i>exp</i> ₂ + <i>exp</i> ₃	if not (<i>typeEqual</i> (<i>exp</i> ₂ .type , <i>integer</i>) and <i>typeEqual</i> (<i>exp</i> ₃ .type , <i>integer</i>)) then <i>type-error</i> (<i>exp</i> ₁) ; <i>exp</i> ₁ .type := <i>integer</i>
<i>exp</i> ₁ <i>exp</i> ₂ or <i>exp</i> ₃	if not (<i>typeEqual</i> (<i>exp</i> ₂ .type , <i>boolean</i>) and <i>typeEqual</i> (<i>exp</i> ₃ .type , <i>boolean</i>)) then <i>type-error</i> (<i>exp</i> ₁); <i>exp</i> ₁ .type := <i>boolean</i>
<i>exp</i> ₁ <i>exp</i> ₂ [<i>exp</i> ₃]	if <i>isArrayType</i> (<i>exp</i> ₂ .type) and <i>typeEqual</i> (<i>exp</i> ₃ .type , <i>integer</i>) then <i>exp</i> ₁ .type := <i>exp</i> ₂ .type.child1 else <i>type-error</i> (<i>exp</i> ₁)
<i>exp</i> num	<i>exp.type</i> := <i>integer</i>
<i>exp</i> true	<i>exp.type</i> := <i>boolean</i>
<i>exp</i> false	<i>exp.type</i> := <i>boolean</i>
<i>exp</i> id	<i>exp.type</i> := <i>lookup</i> (id .name)

假定类型保持某种树形结构，因此在图 6-16 中的文法的一种结构类型 **array** 对应于语义动作

makeTypeNode (*array.size.type*)

构成一个类型节点



这里数组节点的子孙是 *type* 参数给定的类型树。在树的表示中假定简单类型 *integer* 和 *boolean* 构成了标准叶子节点。

2) 语句 语句本身没有类型，但对类型正确性而言需要检查子结构。一般的情形是在示例文法中两个语句规则，*if* 语句和赋值语句。在 *if* 语句的情况中，条件表达式必须是布尔类型。这通过规则

if not typeEqual (exp.type , boolean) then type-error(stmt)

表示，这里 *type-error* 指示一个错误报告机制，其属性将简要地描述。

在赋值语句的情况下，要求被赋值的变量和其接受的值的表达式有相同的类型。这依赖于 *typeEqual* 函数表示的类型等价算法。

3) 表达式 常量表达式，像数字及布尔值 **true** 和 **false**，隐含地说明了 *integer* 和 *boolean* 类型。变量名在符号表中通过 *lookup* 操作确定它们的类型。其他表达式通过操作符构成，如算术操作符 **+**、布尔操作符 **or**、以及下标操作符 **[]**。对每种情况子表达式都必须是指定操作的正确类型。对于下标的情况，这由规则

```

if isArrayType(exp2.type)
  and typeEqual(exp3.type , integer)
  then exp1.type := exp2.type.child1 else type-error(exp1)
  
```

指示这里函数 *isArrayType* 测试其参数是数组类型，即类型的树形表示有一个根节点，表示数组类型构造器。下标表达式导出的类型是数组的基类型，在数组类型的树形表示中它是根节点的 (第一个) 子节点表示的类型，这通过 *exp₂.type.child1* 指示。

现在留下了描述在出现错误时这样的类型检查器的行为，像表 6-10 中语义规则的 *type-error* 过程所指示的那样。主要的问题是何时产生错误消息以及在错误出现时如何继续类型检查。每次出现类型错误时不是都产生错误消息；另一方面，单个的错误可能会引起一连串的许多错误 (有时也恰好出现语法错误)。事实上，如果 *type-error* 过程能确定在有关的位置已经出现了类型错误，那么就可能抑制错误消息的产生。这可以通过一个特别的内部错误类型 (用一空的类型树表示) 发信号。如果在一个子结构中 *type-error* 遇到这个错误类型，就没有错误消息产生。同时，如果错误类型意味着结构的类型不能被确定，那么类型检查器可以当作它的类型 (实际上是未知的) 使用错误类型。例如，在表 6-10 的语义规则中，给定一个下标表达式 *exp₁ exp₂ [exp₃]*，如果 *exp₂* 不是数组类型，那么 *exp₁* 不能被赋予一个有效的类型，并且在语义动作中没有类型赋值。这假定类型域被某个错误类型初始化。另一方面，在操作符 **+** 和 **or** 的情况，即使出现类型错误，这个假设可以使结果意味着整型或布尔型，表 6-10 中的规则也使用它们给结果分配一个类型。

6.4.5 类型检查的其他主题

在这一小节我们简要讨论前面已经讨论过的类型检查算法的一些常用的扩展。

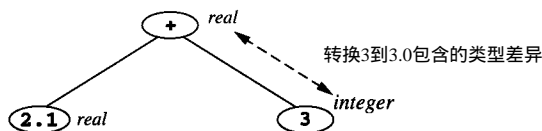
1) 重载 一个操作符是重载的，如果同一操作符名用于了两个不同的操作。重载常用的例

子是算术操作符的情况，通常表示不同数值的操作。例如， $2+3$ 表示整数加，而 $2.1+3.0$ 表示浮点数加，必须通过不同的指令或指令集在内部实现。这样的重载可以扩展到用户说明的函数或过程，相关的操作使用相同的名字，但说明不同的参数或不同的类型。例如，对两个整数和实数值，我们定义取最大值的过程：

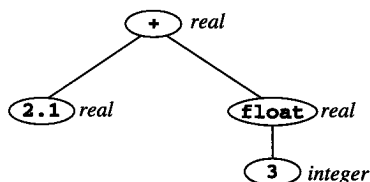
```
procedure max (x,y: integer): integer;
procedure max (x,y: real): real;
```

在Pascal和C中这是非法的，因为它表示了相同的作用域中相同名字的重说明。然而，在Ada和C++中，这样的说明是合法的，因为类型检查器可以根据参数的类型确定要使用哪一个max过程。使用这样的类型检查分清名字的多重含义，可以根据语言的规则用多种方法实现。一种方法用类型参数增加了符号表的lookup过程，允许符号表找到正确的匹配。另一种不同的解决方法是对符号表保持名字所有可能类型的一个集合，并把这个集合返回给类型检查器。这对更复杂的情形是有用的，唯一的类型可以不必立即确定。

2) 类型转换和强制 语言类型规则的一种常用扩充是允许混合类型的算术表达式，如 $2.1+3$ ，一个实数和一个整数相加。在这样的情况下，必须建立一种通用的类型与所有子表达式的类型兼容，在应用操作符之前必须用某种操作把运行的值转换到相应的表示。例如，在表达式 $2.1+3$ 中，在进行加之前整数值3必须转换成浮点数，结果表达式将是浮点数类型。语言进行这样的转换有两种途径。例如，Modula-2要求编程者提供一个转换函数，这样刚才给出的例子可以写成 $2.1+\text{FLOAT}(3)$ ，否则将导致类型错误。另一种可能性(在C语言中使用)是基于子表达式的类型，为类型检查器提供一个自动的转换操作。这样的自动转换称作强制(coercion)。强制能由类型检查器隐含地表示，根据子表达式的类型推断出表达式的类型。如



这要求后面的代码产生器检查表达式的类型确定是否需要应用转换。另一种情况，通过在语法树中插入一个转换节点，类型检查器可以隐含地提供转换，如



类型转换和强制也用于赋值，如

```
r = i;
```

在C语言中，如果r的类型是double，i的类型是int，在存储i为r的值之前，它的值强制为double。这样的赋值在转换期间可能会丢失信息，就像相反方向的赋值(在C中也是合法的)：

```
i = r;
```

类似的情形出现在面向对象的语言中，通常允许子类对象向超类对象赋值(信息也有相应的丢失)。例如，在C++中，如果是A一个类，是B一个子类，并且如果x是A对象，y是B的对象，那么x=y是允许的，但反过来不行(这称作子类型原理(subtype principle))。

3) 多态性类型 有一种语言是多态性(polymorphic)的, 如果允许语言的构造有多种类型。直到现在我们讨论的语言的类型检查本质上是单态 (monomorphic)的, 所有的名字和表达式都要求有唯一的类型。对这种单态性要求的一种放松是重载。但是, 重载通过多态性的一种形式, 只能用于相同名字多种独立说明的情形。当单个说明需要用于任意的类型时就出现了另一种不同的情形。例如, 交换两个变量值的过程在原理上可以用于任何类型的变量 (只要它们类型相同):

```
procedure swap (var x,y: anytype);
```

这个`swap`过程的类型称作被类型 *anytype* 限定(parametrized), *anytype* 被看作是类型变量 (type variable), 能假设成任意实际的类型。可以这样表达这个过程

```
procedure (var anytype, var anytype): void
```

这里*anytype*的每次出现都引用相同的(但是未指定的)类型。这样的类型实际上是类型模式 (type pattern)或类型方案(type scheme)而不是实际的类型, 类型检查器对每次使用 `swap` 的情形都需要确定实际的类型, 匹配这个类型模式或说明一个类型错误。例如, 给定代码

```
var x,y: integer;
    a,b: char;
. . .
swap(x,y);
swap(a,b);
swap(a,x);
```

在调用`swap(x,y)`时, `swap`过程根据其给定的多态性类型模式“指定”到(单态)类型

```
procedure (var integer, var integer): void
```

而在调用`swap(a,b)`时, 它被指定类型

```
procedure (var char, var char): void
```

另一方面, 在调用`swap(a,x)`时, `swap`过程的类型为

```
procedure (var char, var integer): void
```

并且这个类型不能从`swap`的类型模式通过代替类型变量 *anytype* 来产生。存在类型检查算法进行这种一般的多态性类型检查, 特别是在 ML 这样的现代的函数式语言中, 但其中包括复杂的模式匹配技术, 这里不进行研究(参见“注意与参考”一节)。

6.5 TINY语言的语义分析

这一节我们基于前一章构造的 TINY 语法分析程序, 开发 TINY 语言的语义分析程序代码。语义分析程序所基于的 TINY 的语法和语法树结构在 3.7 节描述。

TINY 语言在其静态语义要求方面特别简单, 语义分析程序也将反映这种简单性。在 TINY 中没有明确的说明, 也没有命名的常量、数据类型或过程; 名字只引用变量。变量在使用时隐含地说明, 所有的变量都是整数数据类型。也没有嵌套作用域, 因此变量名在整个程序有相同的含义, 符号表也不需要保存任何作用域信息。

在 TINY 中类型检查也特别简单。只有两种简单类型: 整型和布尔型。仅有的布尔型值是两个整数值的比较的结果。因为没有布尔型操作符或变量, 布尔值只出现在 `if` 或 `repeat` 语句的测试表达式中, 不能作为操作符的操作数或赋值的值。最后, 布尔值不能使用 `write` 语句输出。

我们把对 TINY 语义分析程序的代码的讨论分成两个部分。首先, 讨论符号表的结构及其

相关的操作。然后，语义分析程序自身的操作，包括符号表的构造和类型检查。

6.5.1 TINY的符号表

在TINY语义分析程序符号表的设计中，首先确定什么信息需要在符号表中保存。一般情况这些信息包括数据类型和作用域信息。因为 TINY 没有作用域信息，并且所有的变量都是整型，TINY 符号表不需要保存这些信息。然而，在代码产生期间，变量需要分配存储器地址，并且因为在语法树中没有说明，因此符号表是存储这些地址的逻辑位置。现在，地址可以仅仅看成是整数索引，每次遇到一个新的变量时增加。为使符号表更加有趣和有用，还使用符号表产生一个交叉参考列表，显示被访问变量的行号。

作为符号表产生信息的例子，考虑下列 TINY 程序的例子(加上了行号)：

```

1: { sample program
2:   in TINY language --
3:   computes factorial
4: }
5: read x; { input an integer }
6: if 0 < x then { don compute if x <= 0 }
7:   fact := 1;
8:   repeat
9:     fact := fact * x;
10:    x := x - 1
11:  until x = 0;
12:  write fact { output factorial of x }
13: end

```

这个程序的符号表产生之后，语义分析程序将输出 (TraceAnalyze= True) 下列信息到列出的文件中：

```

Symbol table:
Variable Name      Location      Line      Numbers
-----
x                  0              5          6          9          10         10         11
fact               1              7          9          9          12

```

注意，在符号表中同一行的多次引用产生了那一行的多个入口。

符号表的代码包含在 `syntab.h` 和 `syntab.c` 文件中，在附录 B 中列出 (分别是第 1150 到 1179 行和第 1200 到 1321 行)。

符号表使用的结构是在 6.3.1 节中描述的分离的链式杂凑表，杂凑函数是程序清单 6-2 给出的。因为没有作用域信息，所以不需要 *delete* 操作，*insert* 操作除了标识符之外，也只需要行号和地址参数。需要的其他的两个操作是打印刚才列出的文件中的汇总信息，以及 *lookup* 操作，从符号表中取出地址号 (后面的代码产生器需要，符号表生成器也要检查是否已经看见了变量)。因此，头文件 `syntab.h` 包含下列说明：

```

void st_insert ( char * name, int lineno, int loc );
int st_lookup ( char * name );
void printSymTab(FILE * listing);

```

因为只有一个符号表，它的结构不需要在头文件中说明，也无须作为参数在这些过程中出现。

在 `syntab.c` 中相关的实现代码使用了一个动态分配链表，类型名是 `LineList` (第 1236 行到第 1239 行)，存储记录在杂凑表中每个标识符记录的相关行号。标识符记录本身保存在一个

“桶”列表中，类型名是 `BucketList` (第1247行到第1252行)。`st_insert` 过程在每个“桶”列表 (第1262行到第1295行) 前面增加新的标识符记录，但行号在每个行号列表的尾部增加，以保持行号的顺序 (`st_insert` 的效率可以通过使用环形列表或行号列表的前/后双向指针来改进；参见练习)。

6.5.2 TINY语义分析程序

TINY的静态语义共享标准编程语言的特性，符号表的继承属性，而表达式的数据类型是合成属性。因此，符号表可以通过对语法树的前序遍历建立，类型检查通过后序遍历完成。虽然这两个遍历能容易地组合成一个遍历，为使两个处理步骤操作的不同之处更加清楚，仍把它们分成语法树上两个独立的遍。因此，语义分析程序与编译器其他部分的接口，放在文件 `analyze.h` 中 (附录B，第1350行到第1370行)，由两个过程组成，通过下列说明给出

```
void buildSymtab(TreeNode *);
void typeCheck(TreeNode *);
```

第1个过程完成语法树的前序遍历，当它遇到树中的变量标识符时，调用符号表 `st_insert` 过程。遍历完成后，它调用 `printSymTab` 打印列表文件中存储的信息。第2个过程完成语法树的后序遍历，在计算数据类型时把它们插入到树节点，并把任意的类型检查错误记录到列表文件中。这些过程及其辅助过程的代码包含在 `analyze.c` 文件中 (附录B，第1400行到第1558行)。

为强调标准的树遍历技术，实现 `buildSymtab` 和 `typeCheck` 使用了相同的通用遍历函数 `traverse` (第1420行到第1441行)，它接受两个作为参数的过程 (和语法树)，一个完成每个节点的前序处理，一个进行后序处理：

```
static void traverse ( TreeNode * t,
                      void (* preProc) (TreeNode * ),
                      void (* postProc) (TreeNode * ) )
{ if (t != NULL)
  { preProc(t);
    { int i;
      for (i=0; i < MAXCHILDREN; i++)
        traverse(t->child[i], preProc, postProc);
    }
    postProc(t);
    traverse(t->sibling, preProc, postProc);
  }
}
```

给定这个过程，为得到一次前序遍历，当传递一个“什么都不做”的过程作为 `preproc` 时，需要说明一个过程提供前序处理并把它作为 `preproc` 传递到 `traverse`。对于TINY符号表的情况，前序处理器称作 `insertNode`，因为它完成插入到符号表的操作。“什么都不做”的过程称作 `nullProc`，它用一个空的过程体说明 (第1438行到第1441行)。然后建立符号表的前序遍历由 `buildSymtab` 过程 (第1488行到第1494行) 内的单个调用

```
traverse (syntaxTree, insertNode, nullProc);
```

完成。类似地，`typeCheck` (第1556行到第1558行) 要求的后序遍历由单个调用

```
traverse (syntaxTree, nullProc, checkNode);
```

完成。这里 `checkNode` 是一个适当说明的过程，计算和检查每个节点的类型。现在还剩下描述过程 `insertNode` 和 `checkNode` 的操作。

`insertNode` 过程 (第1447行到第1483行) 必须基于它通过参数 (指向语法树节点的指针) 接受的语法树节点的种类，确定何时把一个标识符 (与行号和地址一起) 插入到符号表中。对于语句节点的情况，包含变量引用的节点是赋值节点和读节点，被赋值或读出的变量名包含在节点的 `attr.name` 字段中。对表达式节点的情况，感兴趣的是标识符节点，名字也存储在 `attr.name` 中。因此，在那3个位置，如果还没有看见变量 `insertNode` 过程包含一个

```
st_insert (t->attr.name, t->lineno, location++);
```

调用 (与行号一起存储和增加地址计数器)，并且如果变量已经在符号表中，则

```
st_insert (t->attr.name, t->lineno, 0);
```

(存储行号但没有地址)。

最后，在符号表建立之后，`buildSymtab` 完成对 `printSymTab` 的调用，在标志 `TraceAnalyze` 的控制下 (在 `main.c` 中设置)，在列表文件中写入行号信息。

类型检查遍的 `checkNode` 过程有两个任务。首先，基于子节点的类型，它必须确定是否出现了类型错误。其次，它必须为当前节点推断一个类型 (如果它有一个类型) 并且在树节点中为这个类型分配一个新的字段。这个字段在 `TreeNode` 中称作 `type` 字段 (在 `globals.h` 中说明，见附录B，第216行)。因为仅有表达式节点有类型，这个类型推断只出现在表达式节点。在 `TINY` 中只有两种类型，整型和布尔型，这些类型在全局说明的枚举类型中说明 (见附录B，第203行)：

```
typedef enum {Void, Integer, Boolean} ExpType;
```

这里类型 `Void` 是“无类型”类型，仅用于初始化和错误检查。当出现一个错误时，`checkNode` 过程调用 `typeError` 过程，基于当前的节点，在列表文件中打印一条错误消息。

还剩下归类 `checkNode` 的动作。对表达式节点，节点可以是叶子节点 (常量或标识符，种类是 `ConstK` 或 `IdK`)，或者是操作符节点 (种类 `OpK`)。对叶子节点的情况 (第1517行到第1520行)，类型总是 `Integer` (没有类型检查发生)。对操作符节点的情况 (第1508行到第1516行)，两个子孙节点表达式的类型必须是 `Integer` (因为后序遍历已经完成，已经计算出它们的类型)。然后，`OpK` 节点的类型从操作符本身确定 (不关心是否出现了类型错误)：如果操作符是一个比较操作符 (<或=)，那么类型是 `Boolean`；否则是 `Integer`。

对语句节点的情况，没有类型推断，但除了一种情况，必须完成某些类型检查。这种情况是 `ReadK` 语句，这里被读出的变量必须自动成为 `Integer` 类型，因此没有必要进行类型检查。所有4种其他语句种类需要一些形式的类型检查：`IfK` 和 `RepeatK` 语句需要检查它们的测试表达式，确保它们是类型 `Boolean` (第1527行到第1530行和第1539行到第1542行)，而 `WriteK` 和 `AssignK` 语句需要检查 (第1531行到第1538行) 确定被写入或赋值的表达式不是布尔型的 (因为变量只能是整型值，只有整型值能被写入)：

```
x := 1 < 2; { error - Boolean value
              cannot be assigned }
write 1 = 2; { also an error }
```

练习

6.1 通过下面文法给出的数的整数值，写出一个属性文法：

```

number    digit number | digit
digit     0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

```

- 6.2 通过下面文法给出的十进制数的浮点数值，写出一个属性文法（提示：使用一个属性 *count* 计算小数点右面数字的个数）。

```

dnum      num.num
num        num digit | digit
digit     0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

```

- 6.3 前一个练习中的十进制数文法可进行重写，不需要属性 *count*（包括它的等式也可避免）。重写文法实现这一点，并为 *dnum* 的值给出一个新的属性文法。
- 6.4 考虑一个表达式文法，它可写成消除左递归的预分析程序：

```

exp       term exp
exp       + term exp | - term exp | ε
term      factor term
term      * factor term | ε
factor    (exp) | number

```

写出用这个文法给出的表达式的值的属性文法。

- 6.5 重写表6-2的属性文法，代替 *val* 计算 *postfix* 串属性，包含简单整数表达式的后缀形式。例如， $(34-3)*42$ 的 *postfix* 属性是 “34 3-42 + *”。可以假设一个串联操作符 *||* 和 *number.strvval* 属性存在。
- 6.6 考虑下面的整数二叉树文法（线性形式）：

```

btree     ( number btree btree ) | nil

```

写出一个属性文法检查二叉树是有序的，即第一个子树中数的值 当前数的值，并且第2个子树所有数的值 当前数的值。例如， $(2 (1 \text{ nil nil}) (3 \text{ nil nil}))$ 是有序的，而 $(1 (2 \text{ nil nil}) (3 \text{ nil nil}))$ 不是。

- 6.7 考虑下面简单的类Pascal 说明的文法：

```

decl      var-list: type
var-list   var-list, id | id
type       integer | real

```

写出变量类型的一个属性文法。

- 6.8 考虑练习6.7的文法。重写这个文法使变量的类型可以说明为纯的合成属性，并给出具有这个特性的类型的新的属性文法。
- 6.9 重写例6.4的文法和属性文法，使 *based-num* 的值能通过单独的合成属性计算。
- 6.10 a. 画出对应于例6.14中每个文法规则的相关图，表达式是 $5/2/2.0$ 。
 b. 描述要求在 $5/2/2.0$ 的语法树上计算属性的两遍，包括节点访问可能的顺序和在每点计算的属性值。
 c. 写出过程的伪代码，完成b中描述的计算。
- 6.11 画出对应于练习6.4中属性文法的每个文法规则的相关图，字符串是 $3*(4+5)*6$ 。
- 6.12 画出对应于练习6.7中属性文法的每个文法规则的相关图，画出说明 *x,y,z:real*

的相关图。

6.13 考虑下面的属性文法：

文法规则	语义规则
$S \rightarrow ABC$	$B.u = S.u$ $A.u = B.v + C.v$ $S.v = A.v$
$A \rightarrow a$	$A.v = 2 * A.u$
$B \rightarrow b$	$B.v = B.u$
$C \rightarrow c$	$C.v = 1$

- 画出字符串 abc 的语法树 (语言仅有的字符串), 画出相关属性的相关图。描述属性等式的正确顺序。
- 假设在属性等式开始前 $S.u$ 赋值为 3。当等式完成时 $S.v$ 的值的多少?
- 假设属性等式修改如下：

文法规则	语义规则
$S \rightarrow ABC$	$B.u = S.u$ $C.u = A.v$ $A.u = B.v + C.v$ $S.v = A.v$
$A \rightarrow a$	$A.v = 2 * A.u$
$B \rightarrow b$	$B.v = B.u$
$C \rightarrow c$	$C.v = C.u - 2$

如果等式开始前 $S.u = 3$, 属性等式完成后 $S.v$ 的值是多少?

6.14 说明对如下给定的属性文法：

文法规则	语义规则
$decl \rightarrow type \text{ var-list}$	$var-list.dtype = type.dtype$
$type \rightarrow \text{int}$	$type.dtype = integer$
$type \rightarrow \text{float}$	$type.dtype = real$
$var-list_1 \rightarrow id, var-list_2$	$id.dtype = var-list_1.dtype$ $var-list_2.dtype = var-list_1.dtype$
$var-list \rightarrow id$	$id.dtype = var-list.dtype$

如果在 LR 分析期间属性 $type.dtype$ 保存在值栈中, 那么当发生 $var-list$ 归约时, 这个值不能在栈中的固定位置找到。

6.15 a. 说明文法 $B \rightarrow ABb \mid a$ 是 SLR(1), 但文法

$$\begin{aligned} B &\rightarrow ABb \mid a \\ A &\rightarrow \epsilon \end{aligned}$$

(由前面文法加上 ϵ 产生式构造), 对任意 k 都不是 LR(k)。

- 给定(a)部分的文法 (带 ϵ -产生式), Yacc 产生的分析程序接受什么字符串?

c. 这种情形与“实际的”编程语言语义分析期间出现的情况是否相似？试说明。

- 6.16 把6.3.5节的表达式文法重写成无二义性文法，用这样的方法那一节写的表达式保持合法性，用这个新文法重写表6-9的属性文法。
- 6.17 使用并列说明代替顺序说明重写表6-9的属性文法。
- 6.18 写一个属性文法，计算6.3.5节中表达式文法的每个表达式的值。
- 6.19 修改程序清单6-6函数`typeEqual`的伪代码，合并类型名并提出确定252页描述的递归类型的结构等价的算法。
- 6.20 考虑下列表达式的(二义)文法：

$$\begin{aligned} \text{exp} \quad & \text{exp} + \text{exp} \mid \text{exp} - \text{exp} \mid \text{exp} * \text{exp} \mid \text{exp} / \text{exp} \\ & \mid (\text{exp}) \mid \text{num} \mid \text{num}.\text{num} \end{aligned}$$

假设在计算任何这样的表达式时遵循C语言的规则：如果两个表达式是混合类型的，那么整型的子表达式转换成浮点型，并应用浮点型操作符。写一个属性文法把这样的表达式转换成在Modula-2中也是合法的表达式：从整数到浮点数的转换使用FLOAT函数表达，如果两个操作数都是整数，除法操作符/就被视为div。

- 6.21 考虑对图6-16的下列文法的扩展，它包括了函数说明和调用：

```

program    var-decls ; fun-decls ; stmts
var-decls  var-decls ; var-decl | var-decl
var-decl   id : type-exp
type-exp   int | bool | array [num] of type-exp
fun-decls  fun id ( var-decls ) : type-exp ; body
body       exp
stmts      stmts ; stmt | stmt
stmt       if exp then stmt | id := exp
exp        exp + exp | exp or exp | exp [exp] | id ( exps )
           | num | true | false | id
exps       exps , exp | exp

```

- a. 为新的函数类型结构设计一个合适的树结构，为两个函数类型写一个 `typeEqual` 函数。
- b. 写出函数说明和函数调用类型检查的语义规则(由规则 `exp id (exp)` 表示)，类似于表6-10的规则。
- 6.22 考虑下列C表达式的二义文法。给定表达式

`(A) - x`

如果`x`是一个整型变量，`A`在`typedef`中说明等价于`double`，那么这个表达式计算`-x`的值为`double`类型。另一方面，如果`A`是一个整型变量，则计算两个变量的整型差值。

- a. 描述分析程序如何使用符号表区分这两种解释。
- b. 描述扫描器如何使用符号表区分这两种解释。
- 6.23 对应于TINY类型检查器的强制类型约束写一个属性文法。

6.24 写出一个TINY语义分析程序符号表构造的属性文法。

编程练习

6.25 写出例6.4基数语法树的C语言说明, 并使用这些说明把例6.13的*EvalWithBase*伪代码转换成C代码。

6.26 a. 重写程序清单4-1的递归下降求值程序, 代替表达式的值而打印出后缀转换式(见练习6.5)。

b. 重写递归下降求值程序, 打印出值和后缀转换式。

6.27 a. 为一个简单整数计算重写程序清单5-1的Yacc规范, 代替表达式的值而打印出后缀转换式(见练习6.5)。

b. 重写Yacc规范, 打印出值和后缀转换式。

6.28 写出一个Yacc规范, 打印出练习6.20中文法给出的表达式的Modula-2转换式。

6.29 写出一个程序的Yacc规范, 用于计算带let-块的表达式的值(表6-9)(可以把let和in记号缩写成一个字符, 并约束标识符或使用Lex产生一个合适的扫描器)。

6.30 写出一个程序的Yacc和Lex规范, 进行语言的类型检查, 其文法在图6-16给出。

6.31 重写TINY语义分析程序符号表的实现, 数据结构 `LineList` 加进一个向后的指针, 并提高insert操作的效率。

6.32 重写TINY语义分析程序, 使其只对语法树进行一遍遍历。

6.33 TINY语义分析程序在变量使用之前, 没有确保其已被赋值。因此, 下面的 TINY代码在语义上认为是正确的:

```
y := 2+x;  
x := 3;
```

重写TINY分析程序进行“合理的”检查, 在表达式中一个变量的赋值发生在使用之前。什么妨碍这样的检查十分简单?

6.34 a. 重写TINY语义分析程序, 允许布尔值存储到变量中。这将要求在符号表中给定变量的数据类型为布尔型或整型。TINY程序的类型正确性现在必须包括对变量所有的赋值(和使用)与它们的数据类型一致的要求。

b. 根据a中的修改, 写出一个TINY类型检查器的属性文法。

注意与参考

属性文法的早期工作主要是 Knuth[1968]进行的。在编译器构造中使用属性文法的进一步研究出现在 Lorho[1984]。正式使用属性文法指定编程语言的语义是 Slonneger和Kurtz[1995]的研究, 这里为类似于TINY的语言的静态语义给出了一个完整的属性文法。属性文法的其他数学特性可以在 Mayoh[1981]中找到。一种非闭环的测试可以在 Jazayeri、Ogden和Rounds[1975]中找到。

分析期间属性的赋值问题在 Fischer和LeBlanc[1991]的研究中更加详细一些。在LR分析期间确保其进行的条件在 Jones[1980]中。在调度动作中加进 ϵ -产生式(像在Yacc中)保持确定性的LR分析的问题在Purdum和Brown中研究。

符号表实现的数据结构, 包括杂凑表及其效率分析, 能在许多文章中找到; 例如参见 Aho、

Hopcroft和Ullman[1983]或Cormen、Leiserson和Rivest[1990]。选择杂凑函数的细致的研究在Knuth[1973]中。

类型系统、类型正确性和类型推断形成了理论计算机科学研究的一个主要的领域，并应用到许多语言中。通常的概要参见 Louden[1993]。更进一步的观点参见 Cardelli和Wegner[1985]。C和Ada使用类型等价的混合形式，类似于 Pascal的等价说明，很难简洁地描述。较早的语言，如FORTRAN77、Algol60和Algol68使用结构等价。像ML和Haskell使用严格的名等价，用类型同义词代替结构等价。在 6.4.3节中描述的结构等价算法可以在 Koster[1969]中找到；类似算法的现代的应用在 Amadio和Cardelli[1993]中。多态的类型系统和类型推导算法在 Peyton Jones[1987]和Reade[1989]中描述。在ML和Haskell中使用的多态的类型推导称作 **Hindley-Milner**类型推导[Hindley, 1969; Milner, 1978]。

本章中我们没有描述任何属性求值的自动构造工具，因为通常并不使用（不像扫描器和分析程序产生器 Lex和Yacc）。基于属性文法的一些有趣的工具是 LINGUIST[Farrow, 1984]和 GAG[Kastens, Hutt和Zimmermann, 1982]。合成器产生器[Reps和Teitelbaum, 1989]是一个成熟的工具，用于基于属性文法的上下文有关编辑器的产生。对这个工具可做的一件有趣的事是构造一个语言编辑器，自动地提供基于使用的变量声明。

China-pub.com

下载

第7章 运行时环境

本章要点

- 程序执行时的存储器组织
- 完全静态运行时环境
- 基于栈的运行时环境
- 动态存储
- 参数传递机制
- TINY 语言的运行时环境

在前几章中，我们已研究了实现源语言静态分析的编译程序各阶段。该内容包括了扫描、分析和静态语义分析。这个分析仅仅取决于源语言的特性，它与目标（汇编或机器）语言及目标机器和它的操作系统的特性完全无关。

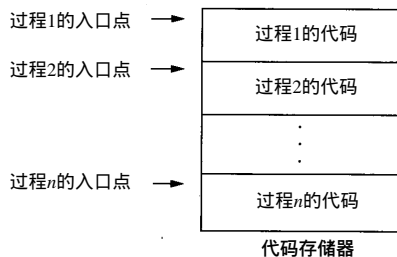
在本章及下一章中，我们将转向研究编译程序如何生成可执行代码的问题。这个研究包括了附加分析，例如由优化程序实现的分析，其中的一些可以与机器无关。但是代码生成的许多任务都依赖于具体的目标机器。然而同样地代码生成的一般特征在体系结构上仍保留了很大的变化。运行时环境(runtime environment)尤为如此，运行时环境指的是目标计算机的寄存器以及存储器的结构，用来管理存储器并保存指导执行过程所需的信息。实际上，几乎所有的程序设计语言都使用运行时环境的3个类型中的某一个，它的主要结构并不依赖于目标机器的特定细节。环境的这3个类型分别是：FORTRAN77的完全静态环境(fully static environment)特征、像C、C++、Pascal以及Ada这些语言的基于栈的环境(stack-based environment)，以及像LISP这样的函数语言的完全动态环境(fully dynamic environment)。这3种类型的混合形式也是可能的。

本章将按顺序逐个讨论这3种环境，还指出哪些环境是可行的语言特征以及它们必须具有的特性。这包括了作用域及分配问题、过程调用的本质和不同的参数传递机制。这一章集中讨论的是环境的一般结构，而第8章着重于维护环境需要生成的真实代码。在这一点上，大家应记住编译程序只能间接地维护环境，在程序执行期间它必须生成代码进行必要的维护操作。相反地由于解释程序可以在其自己的数据结构中直接维护环境，因而它的任务就很简单。

本章的第一节包括了对所有运行时环境的一般特征及其与目标机器的体系结构之间的关系论述；之后的两节探讨了静态环境和基于栈的环境，以及执行时的操作示例。由于基于栈的环境是最常见的，所以我们对于基于栈系统的不同变型和结构又要着重讲述。在这之后是一些动态存储问题，其中包括了完全动态环境和面向对象的环境。下面还会讲到有关环境操作的各种参数传递技术。本章最后简要描述了实现TINY语言所需的简单环境。

7.1 程序执行时的存储器组织

典型计算机的存储器可分为寄存器区域和较慢的直接编址的随机访问存储器（RAM）。RAM区域还可再分为代码区 and 数据区。在绝大多数的语言中，执行时不可能改变代码区，且在概念上可将代码和数据区看作是独立的。另外由于代码区在执行之前是固定，所以在编译时所有代码的地址都是可计算的，代码区可如下所示：



特别地，在编译时还可以知道每个过程的入口点和函数^①。对数据的分配不能这样说，它只有一小部分可在执行之前被分配到存储器中的固定位置。本章大部分内容都会谈论如何处理非固定的或动态的数据分配。

在执行之前，可以将一类数据固定在存储器中，它还包括了程序的全局和 / 或静态数据 (FORTRAN77 与绝大多数的语言不同，它所有的数据都属于这一类)。这些数据通常都在一个固定区域内并以相似的风格单独分配给代码。在 Pascal 中，全局变量属于这一类，C 的外部静态变量也是如此。

在组织全局/静态区域中出现的一个问题是它涉及到编译时所知的常量。这其中包括了 C 和 Pascal 的 `const` 声明以及代码本身所用的文字值，例如串 “Hello %d\n” 和在 C 语句

```
printf ( " Hello %d\n ", 12345 );
```

中的整型值 12345。诸如 0 和 1 这样较小的编译时常量通常由编译程序直接插入到代码中且不为其分配任何数据空间。同样地，由于编译程序已掌握了全局函数或过程的入口点且可直接将其插入到代码中，所以也不为它们分配全局数据区。然而我们却将大型的整型值、浮点值，特别是串文字分配到全局/静态区域中的存储器，在启动时仅保存一次，之后再由执行代码从这些位置中得到(实际上，在 C 中串文字被看作是指针，因此它们必须按照这种方式来保存)。

用作动态数据分配的存储区可按多种方式组织。典型的组织是将这个存储器分为栈 (stack) 区域和堆(heap)区域，栈区域用于其分配发生在后进先出 LIFO(last-in, first-out)风格中的数据，而堆区域则用于不符合 LIFO 协议(例如在 C 中的指针分配)的动态分配^②。目标机器的体系结构通常包括处理器栈，利用了这个栈使得用处理器支持过程调用和返回(使用基于存储器分配的主要机制)成为可能。有时，编译程序不得不将处理器栈的显式分配安排在存储器内的恰当位置中。

一种一般的运行时存储器组织如下所示，它具有上述所有的存储器分类：



① 更为可能的情况是，代码由装载程序装载到存储器的一个在执行开始时分配的区域中，因此这是完全不可预测的。但是之后的所有实际地址都由从固定的装载基地址的偏移自动计算得出，因此和固定地址的原理相同。有时，编译器的编写者必须留心生成可重定位代码 (relocatable code)，其中都相对于某个基址 (base) (通常是寄存器) 执行转移、调用以及引用。下一章将给出一些例子。

② 读者应注意到堆通常是一个简单的线性存储器区域。将它称之为堆是一个历史原因，这与算法 (如堆类排序) 中用到的堆数据结构无关。

上图中的箭头表示栈和堆的生长方向。传统上是将栈画作在存储器中向下生长,这样它的顶部实际就是在其所画区域的底部。堆也画得与栈相似,但它不是 LIFO 结构且它的生长和缩短比箭头所表示的还要复杂(参见 7.4 节)。在某些组织中,栈和堆被分配在不同的存储器部分,而不是占据相同的区域。

存储器分配中的一个重要单元是过程活动记录(procedure activation record),当调用或激活过程或函数时,它包含了为其局部数据分配的存储器。活动记录至少应包括以下几个部分:

自变量(参数)空间
用作簿记信息的空间,它包括了返回地址
用作局部数据的空间
用作局部临时变量的空间

在这里应强调(而且以后还要重复)这个图示仅仅表示的是活动记录的一般组织。包括其所含数据的顺序的特定细节则依赖于目标机器的体系结构、被编译的语言特性,甚至还有编译程序的编写者的喜好。

所有过程活动记录的某些部分,例如用于簿记信息的空间,具有相同的大小。而其他部分,诸如用于自变量和局部数据的空间会对每一个过程保持固定,但是每个过程都各不相同。某些活动记录还会由处理器自动分配到过程调用上(例如存储返回地址)。其他部分(如局部临时变量空间)可能需要由编译程序生成的指令显式地分配。根据语言的不同,可能将活动记录分配在静态区域(FORTRAN77)、栈区域(C、Pascal)、或堆区域(LISP)。当将活动记录保存在栈中时,它们有时指的是栈框架(stack frame)。

处理器寄存器也是运行时环境的结构部分。寄存器可用来保存临时变量、局部变量甚至是全局变量。当处理器具有多个寄存器时,正如在较新的 RISC 处理器中一样,整个静态区域和整个活动记录都可完整地保存在寄存器中。处理器还具有特殊用途的寄存器以记录执行,如在大多数的体系结构中的程序计数器(pc)、栈指针(sp)(stack pointer)。可能还会为跟踪过程活动而特别设计寄存器。这样的寄存器典型的有指向当前活动记录的框架指针(fp)(frame pointer),以及指向保存自变量(参数值)的活动记录区域的自变量指针(argument pointer)[⊖]。

运行时环境的一个特别重要的部分是当调用过程或函数时,对必须发生的操作序列的判定。这样的操作可能还包括活动记录的存储器分配、计算和保存自变量以及为了使调用有效而进行的必要的寄存器的保存和设置。这些操作通常指的是调用序列(calling sequence)。过程或函数返回时需要的额外操作,如放置可由调用程序访问的返回值、寄存器的重新调整,以及活动记录存储器的释放,也通常被认为是调用序列的一个部分。如果需要,可将调用时执行的调用序列部分称作是调用序列(call sequence),而返回时执行的部分称为返回序列(return sequence)。

调用序列设计的重要方面有:1)如何在调用程序和被调用程序之间分开调用序列操作(也就是有多少调用序列的代码放在调用点上,多少放在每个过程的代码开头);2)在多大程度上依赖处理器对调用支持而不是为调用序列的每一步生成显式代码。由于在调用点上比在被调用

⊖ 这些名称都是从 VAX 体系结构中得到的,但是类似的名称还可出现在其他体系结构中。

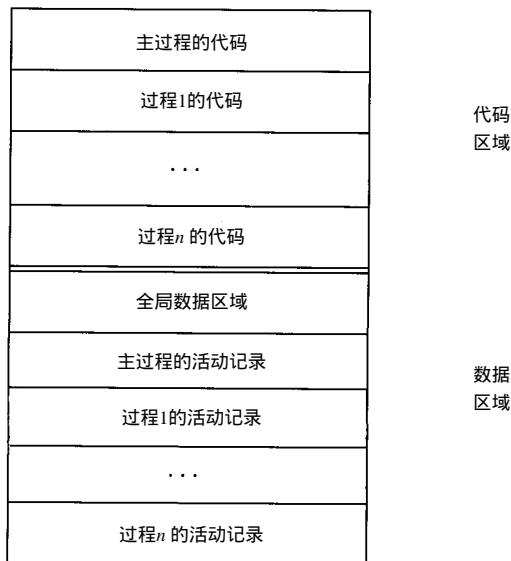
程序内更易生成调用序列代码，但是这样做会引起生成代码数量的生长，因为在每个调用点都要复制相同的代码所以第1点是一个很棘手的问题，后面还要再更详细地讲到这些问题。

调用程序至少要负责计算自变量并将它们放在可由被调用程序找到的位置上（可能是直接放在被调用程序的活动记录中）。此外，调用程序或被调用程序或两者必须保存调用点上的机器状态，包括返回地址，可能还有正使用的寄存器。最后，在调用程序和被调用程序之间须用某个可能的合作方式建立所有附加的簿记信息。

7.2 完全静态运行时环境

最简单的运行时环境类型是所有数据都是静态的，且执行程序期间在存储器中保持固定。这样的环境可用来实现没有指针或动态分配，且过程不可递归调用的语言。此类语言的标准例子是FORTRAN77。

在完全静态环境中，不仅全局变量，所有的变量都是静态分配。因此，每个过程只有一个在执行之前被静态分配的活动记录。我们都可通过固定的地址直接访问所有的变量，而不论它们是局部的还是全局的，则整个程序存储器如下所示：



在这样的环境中，保留每个活动记录的簿记信息开销相对较小，而且也不需要活动记录中保存有关环境的额外信息（而不是返回地址）。用于这样环境的调用序列也十分简单。当调用一个过程时，就计算每个自变量，并将其保存到被调用过程的活动中恰当的参数位置。接着保存调用程序代码中的返回地址，并转移到被调用的过程的代码开头。返回时，转移到返回地址^①。

例7.1 作为这种环境的具体示例，考虑程序清单 7-1中的FORTRAN77程序。这个程序有一个主过程和一个附加的过程QUADMEAN^②。在主过程和QUADMEAN中有由COMMON MAXSIZE声明

① 在大多数的体系结构中，子例程转移自动地保存返回地址；当执行返回指令时，也自动地再装载这个地址。

② 我们忽略库函数SQRT，它由QUADMEAN调用而且在执行之前先被链接。

给出的全局变量^①。

程序清单7-1 一个FORTRAN77示例程序

```

PROGRAM TEST
COMMON MAXSIZE
INTEGER MAXSIZE
REAL TABLE(10),TEMP
MAXSIZE = 10
READ *, TABLE(1),TABLE(2),TABLE(3)
CALL QUADMEAN(TABLE,3,TEMP)
PRINT *, TEMP
END

SUBROUTINE QUADMEAN(A,SIZE,QMEAN)
COMMON MAXSIZE
INTEGER MAXSIZE,SIZE
REAL A(SIZE),QMEAN, TEMP
INTEGER K
TEMP = 0.0
IF ((SIZE.GT.MAXSIZE).OR.(SIZE.LT.1)) GOTO 99
DO 10 K = 1,SIZE
    TEMP = TEMP + A(K)*A(K)
10 CONTINUE
99 QMEAN = SQRT(TEMP/SIZE)
RETURN
END

```

忽略存储器中整型值和浮点值间可能有的大小区别，我们显示了图7-1中的这个程序的运行时环境^②。在该图中，我们用箭头表示从主过程中调用时，过程 QUADMEAN 的参数 A、SIZE 和 QMEAN 的值。在 FORTRAN77 中，参数值是隐含的存储引用，所以调用 (TABLE、3 和 TEMP) 的参数地址就被复制到 QUADMEAN 的参数地址中。它有几个后果。首先，需要一个额外的复引用来访问参数值。其次，数组参数无需再重新设置和复制(因此，只给在 QUADMEAN 中的数组参数 A 分配一个空间，在调用时指出 TABLE 的基地址)。再次，像在调用中的值 3 的常量参数必须被放在一个存储器地址中而且在调用时要使用这个地址(7.5 节将更完整地讨论参数传递机制)。

图7-1中还有一个特性需要解释一下，这

全局区

主过程的
活动记录

过程QUADMEAN
的活动记录

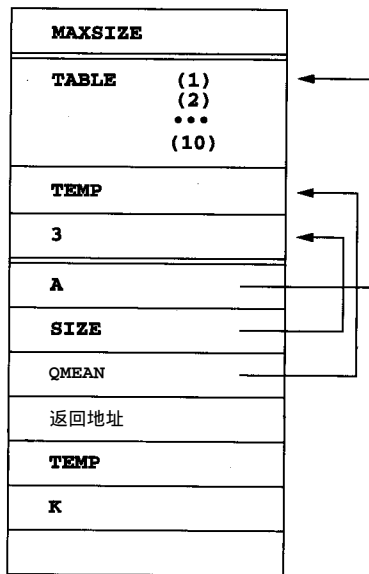


图7-1 程序清单7-1中程序的运行时环境

- ① 实际上，FORTRAN77允许COMMON变量在不同的过程中具有不同的名称，而仍旧指的是相同存储器位置。从这个示例开始，我们将默认忽略这种复杂性。
- ② 我们再次强调这个图示仅仅是示意性的。在操作中的实现实际与这里给出的有所差异。

就是QUADMEAN的活动记录结尾分配的未命名的地址。这个地址是一个在算术表达式计算中用来储存临时变量值的“凑合的”地址。在QUADMEAN中可能需要两个运算。一个是循环中的 $TEMP + A(K) * A(K)$ 的计算,另一个是当参数在调用SQRT时的 $TEMP / SIZE$ 的计算。我们早已谈过需要为参数值分配空间(尽管在对库函数的调用中实际上是有差别的)。循环计算中也需要临时变量存储地址的原因在于每个算术运算必须在一个步骤中,所以就计算 $A(K) * A(K)$ 并在下一步中添加TEMP的值。如果没有足够的寄存器来放置这个临时变量值,或如果有一个调用要求保存这个值,那么在完成计算之前应先将值存储在活动记录中。编译程序可以总是先预测出在执行中它是否必要,然后再为分配临时变量的地址的恰当数量(以及大小)作出安排。

7.3 基于栈的运行时环境

在允许递归调用以及每一个调用中都重新分配局部变量的语言中,不能静态地分配活动记录。相反地,必须以一个基于栈的风格来分配活动记录,即当进行一个新的过程调用(活动记录的压入(push))时,每个新的活动记录都分配在栈的顶部,而当调用退出时则再次解除分配(活动记录的弹出(pop))。活动记录的栈(stack of activation record)(也指运行时栈(runtime stack)或调用栈(call stack))就随着程序执行时发生的调用链生长或缩小。每个过程每次在调用栈上可以有若干个不同的活动记录,每个都代表了一个不同的调用。这样的环境要求的簿记和变量访问的技术比完全静态环境要复杂许多。特别地,活动记录中必须有额外的簿记信息,而且调用序列还包括设置和保存这个额外信息所需的步骤。基于栈的环境的正确性和所需簿记信息的数量在很大程度上依赖于被编译的语言的特性。在本节中为了提高难度复杂性,我们将考虑基于栈的环境的组织,它是由所涉及到的语言特性区分的。

7.3.1 没有局部过程的基于栈的环境

在一个所有过程都是全局的语言(例如C语言)中,基于栈的环境有两个要求:指向当前活动记录的指针的维护允许访问局部变量,以及位置记录或紧前面的活动记录(调用程序的活动记录)允许在当前调用结束时恢复活动记录(且舍掉当前活动)。指向当前活动的指针通常称为框架指针(frame pointer)或fp,且通常保存在寄存器中(通常也称作fp)。作为一个指向先前活动记录的指针,有关先前活动的信息一般是放在当前活动中,并被认为是控制链(control link)或动态链(dynamic link)(之所以称之为动态的,是因为在执行时它指向调用程序的活动记录)。有时将这个指针称为旧fp(old fp),这是因为它代表了fp的先前值。通常,这个指针被放在栈中参数区域和局部变量区域之间的某处,并且指向先前活动记录控制链。此外,还有一个栈指针(stack pointer)或sp,它通常指向调用栈上的最后位置(它有时称作栈顶部(top of stack)指针,或tos)。

现在考虑几个例子。

例7.2 利用Euclid算法的简单递归实现,计算两个非负整数的最大公约数,它的代码(C语言)在程序清单7-2中。

程序清单7-2 例7-2的C代码

```
#include <stdio.h>

int x,y;
```

```

int gcd( int u, int v)
{ if (v == 0) return u;
  else return gcd(v,u % v);
}

main()
{ scanf("%d%d",&x,&y);
  printf("%d\n",gcd(x,y));
  return 0;
}

```

假设用户在该程序中输入了值 15 和 10，那么 main 就初始化调用 gcd(15,10)。这个调用导致了另一个递归调用 gcd(10,5)，(因为 $15 \% 10 = 5$)，而这又引起了第 3 个调用 gcd(5,0)(因为 $10 \% 5 = 0$)，它将返回值 5。在第 3 个调用中，运行时环境如图 7-2 所示。请读者注意指向每个调用的 gcd 是如何向栈的顶部添加新的大小完全相同的活动记录，并且在每个新活动记录中，控制链指向先前活动记录的控制链。还请大家注意，fp 指向当前活动记录的控制链，因此在下一个调用中当前的 fp 就会变成下一个活动记录的控制链了。

调用最后一个 gcd 的之后，将按顺序从栈中删去每个活动，这样当在 main 中执行 printf 语句时，只在环境中保留了 main 和全局/静态区域的活动记录(我们已将 main 的记录显示为空。在实际中，它应包含将控制传回到操作系统的信息)。

最后，应指出在调用 gcd 时调用程序不需要为自变量值安排空间(与图 7-1 中的 FORTRAN77 环境中的常量 3 不同)，这是因为 C 语言使用值参数。7.5 节将详细探讨这一点。

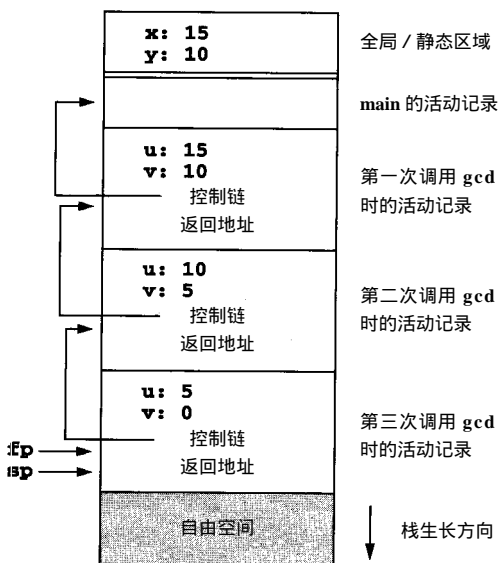


图7-2 例7.2的基于栈的环境

例7.3 考虑程序清单 7-3 中的 C 代码。这个代码包括将用来进一步描述本节相关内容的变量，但是它的基本操作如下所示。从 main 来的第 1 个调用是到 g(2) 的(这是因为 x 在这一点上有值 2)。在这个调用中，m 变成了 2，而 y 则变成了 1。接着 g 调用 f(1)，而 f 也相应地调用 g(1)。在这个到 g 的调用中，m 变成了 1，而 y 则变成了 0，所以再也没有别的调用了。该点(在对 g 的第二次调用期间)上的运行时环境显示在图 7-3a 中。

程序清单 7-3 例7.3的C程序

```

int x = 2;

void g(int); /* prototype */

void f(int n)
{ static int x = 1;

```

```

g(n);
x--;
}

void g(int m)
{ int y = m--1;
  if (y > 0)
  { f(y);
    x--;
    g(y);
  }
}

main()
{ g(x);
  return 0;
}

```

现在对 g 和 f 的调用退出(f 在返回之前它的静态局部变量 x 减1)，它们的活动记录从栈中弹出，且控制返回到紧随在第1次对 g 的调用的 f 的调用之后的点。现在 g 给外部变量 x 减1，并进行另一次调用 $g(1)$ ，将 m 设为2将 y 设为1，这样就得到了图7-3b中的运行时环境。在此之后再也没有调用了，从栈中就会弹出剩余的活动记录，程序退出。

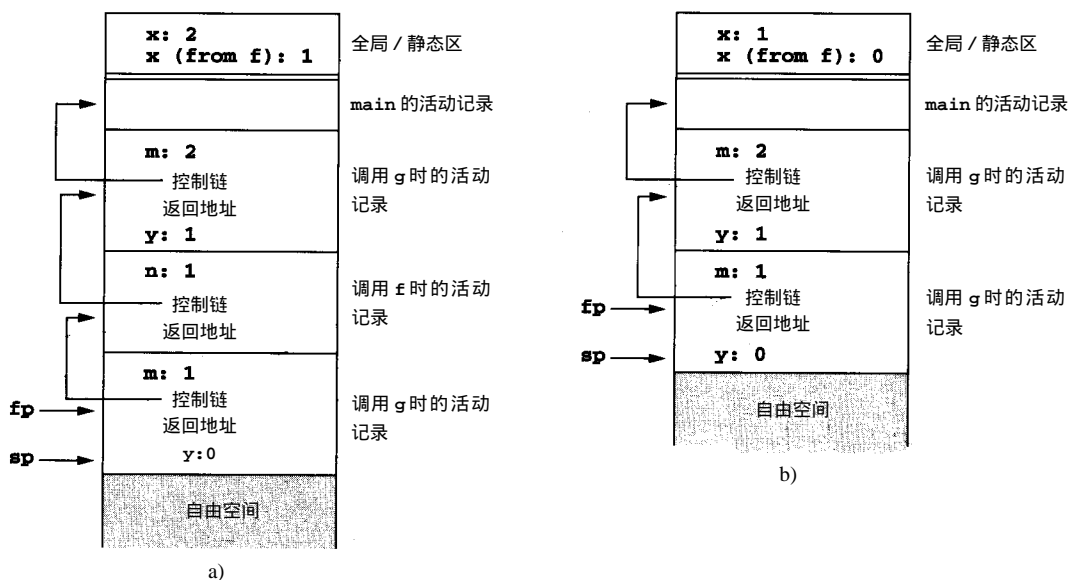


图7-3 程序清单7-3中程序的运行时环境

a) 在第2次对 g 的调用时，程序清单7-3中程序的运行时环境

b) 在第3次对 g 的调用时，程序清单7-3中程序的运行时环境

请注意，在图7-3b中，对 g 的第3次调用的活动记录占据着(并覆盖) f 的活动记录先前占着的存储器区域。大家还应注意：由于 f 中的静态变量 x 必须坚持通过所有对 f 的调用，所以不可在 f 的活动记录中分配。因此，尽管不是全局变量，也必须在全局/静态区域中与外部变量 x

在一起分配。因为符号表会总能区分它与外部的 x ，并在程序每一个点上判定访问的正确变量，所以它们不会有什么混淆。

活动树(activation tree) 是分析程序中复杂调用结构的有用工具，每个活动记录(或调用)都成为该树的一个节点，而每个节点的子孙则代表了与该节点相对应的调用时进行的调用。例如，程序清单 7-2 中程序的活动树是线性的，在图 7-4a 中描述(对于输入 15 和 10 而言)，而程序清单 7-3 中程序的活动树在图 7-4b 中描述。请注意，图 7-2 和图 7-3 中显示的环境表示了调用时由活动树的每个叶子代表的调用。一般而言，在特定调用开头的活动记录栈与活动树的相应节点到根节点的通路有一个结构等价。

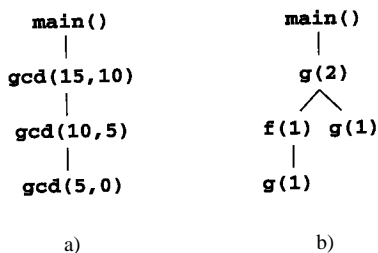
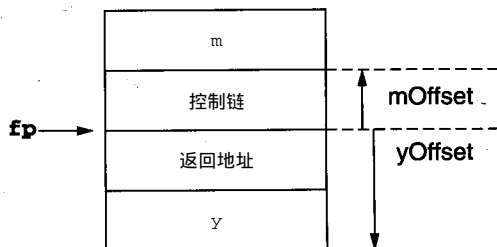


图7-4 程序清单7-2和程序清单7-3中程序的活动树

1) 对名称的访问 在基于栈的环境中，再也不能像在完全静态环境中那样用固定的地址访问参数和局部变量。而它们由当前框架指针的偏移量发现。在大多数的语言中，每个局部声明的偏移量仍是可由编译程序静态地计算出来，因为过程的声明在编译时是固定的，而且为每个声明分配的存储器大小也根据其数据类型而固定。

考虑程序清单 7-3 中 C 程序的过程 g (参见图 7-3 中画出的运行时环境)。 g 的每个活动记录的格式完全相同，而参数 m 和局部变量 y 也总是位于活动记录中完全相同的相对位置。我们把这个距离称作 $mOffset$ 和 $yOffset$ 。然后，在对 g 的任何调用期间，都有下面的局部环境图：



m 和 y 都可根据它们从 fp 的固定偏移进行访问。例如，具体地假设运行时栈从存储器地址的高端向低端生长，整型变量的存储要求两个字节，地址要求 4 个字节。若活动记录的组织如上所画，就有 $mOffset = +4$ 和 $yOffset = -6$ ，且对 m 和 y 的引用写成机器代码(假设是标准的汇编器约定)为 $4(fp)$ 和 $-6(fp)$ 。

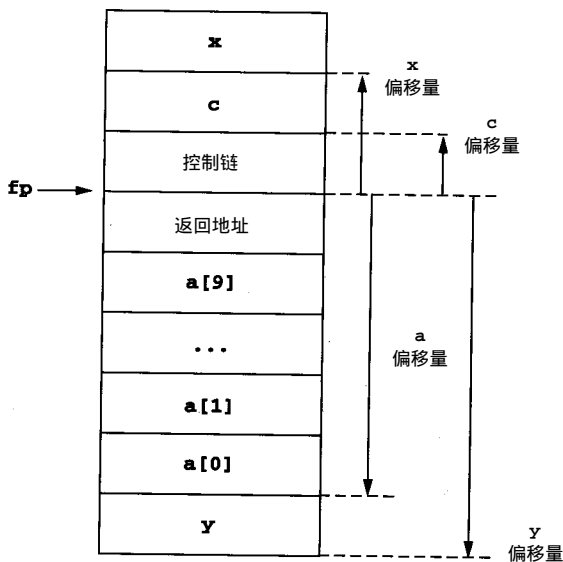
局部数组和结构与简单变量相比分配和计算地址并不更加困难，如下示例所示。

例7.4 考虑C过程

```

void f(int x, char c)
{ int a[10];
  double y;
  ...
}
  
```

对 f 调用的活动记录显示为：



而且假设整型是两个字节、地址 4 个字节、字符 1 个字节、双精度浮点数 8 个字节，那么就有以下的偏移值(再次假设栈的生成为负方向)，这些值在编译时都是可计算的：

名 称	偏 移 量
x	+5
c	+4
A	-24
y	-32

现在对 $a[i]$ 一个访问将要求计算地址

$(-24+2*i)(fp)$

(这里在产生式 $2*i$ 中的因子是比例因子(scale factor)，它是从假设整型值占有两个字节得来的)。这样的存储器访问依赖于 i 的地址以及体系结构，可能只需要一条指令。

对于这个环境中的非局部的和静态名字而言，不能用同局部名字一样的方法访问它们。实际上，我们此处所考虑的情况——不带有过程的语言——所有的非局部的名字都是全局的，因此也就是静态的。所以在图 7-3 中，外部的(全局的)C 变量 x 具有一个固定的静态地址，因此也就可以被直接访问(或是通过某个基指针的偏移而不是 fp)。对来自 f 的静态局部变量 x 的访问也使用完全相同的风格。请注意，正如前一章所描述的，这个机制实现静态(或词法的)作用域。如果需要动态作用域，那么就要使用一个更为复杂的访问机制(本节后面将要提到)。

2) 调用序列 调用序列大致由以下的步骤组成^①。当调用一个过程时，

计算自变量并将其存放在过程的新活动记录中的正确位置(将其妥当地压入到运行时栈中就可做到这一点)。

将 fp 作为控制链存放(压入)到新的活动记录中。

改变 fp 以使其指向新的活动记录的开始(如已有了一个 sp ，则将该 sp 复制到该点上的 fp 中，也可以做到这一点)。

^① 这个描述忽略了必须发生的寄存器的任何保存。它还忽略了将返回值放在一个可用的地址的需要。

将返回地址存放在新的活动记录中(如果需要)。

完成到被调用的过程的代码一个转移。

当存在着一个过程时，则

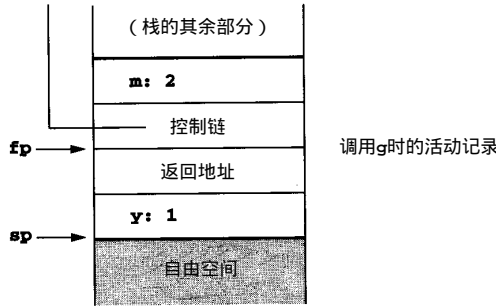
将fp复制到sp中。

将控制链装载到fp中。

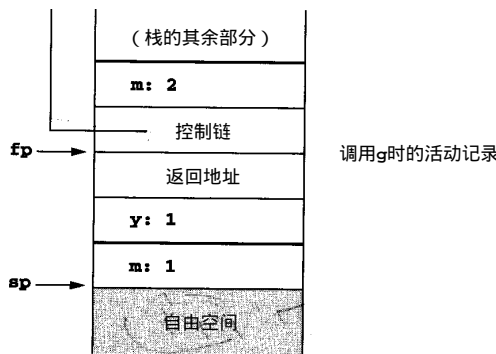
完成到返回地址的一个转移。

改变sp以弹出自变量。

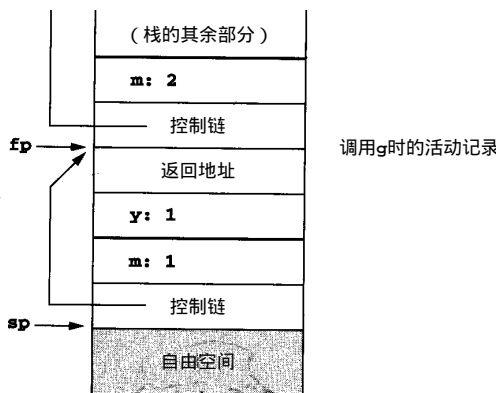
例7.5 考虑在前面图7-3b 中的对g的最后一个调用的情况：



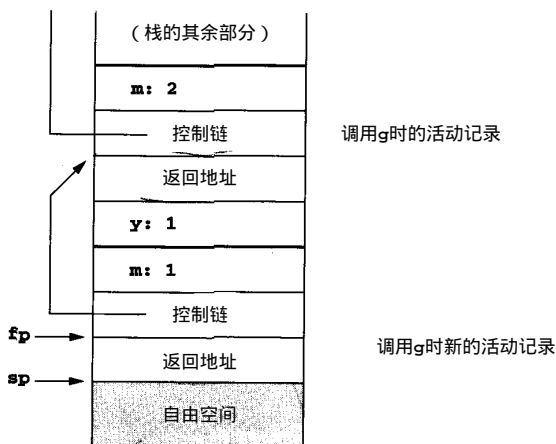
当进行对g新的调用时，首先将参数m的值压入到运行时栈中：



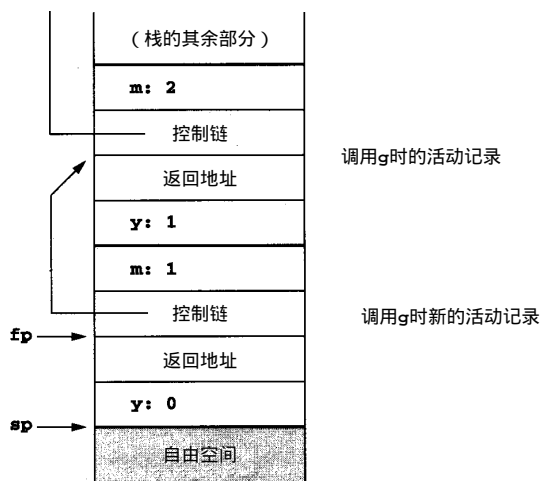
接着将fp压入到栈中：



现在将sp复制到fp中，并将返回的地址压入到栈中，就得出了到g的新的调用了：



最后，g在栈上分配和初始化新的y以完成对新的活动记录的构造：



3) 处理可变长度数据 到这里我们已经描述了一种情况，所有的数据，无论是局部的还是全局的，都可在一个固定的地方，或由编译程序计算出的到fp的固定偏移处找到。有时编译程序必须处理数据变化的可能性，表现在数据对象的数量和每个对象的大小上。发生在支持基于栈的环境的语言中的两个示例如下：调用中的自变量的数量可根据调用的不同而不同。数组参数或局部数组变量的大小可根据调用的不同而不同。

情况(1)的典型例子是C中的printf函数，其中的自变量的个数由作为第一个自变量传递的格式串决定。因此：

```
printf("%d%s%c", n, prompt, ch);
```

就有4个自变量(包括格式串"%d%s%c")，但是

```
printf("Hello, world\n");
```

却只有一个自变量。通常，C编译程序一般通过把调用的自变量按相反顺序 (in reverse order) 压入到运行时栈p来处理这一点。那么，在上面描述的实现中第1个参数(它告知printf的代码

共有多少参数)通常是位于到 fp 的固定偏移的位置(使用上一个示例的假设得出实际上是 +4)。另一个选择是使用一个在 VAX 体系结构中的诸如 ap (自变量指针) 的处理器机制。还会对这以及其他的可能情况在练习中再进一步讨论。

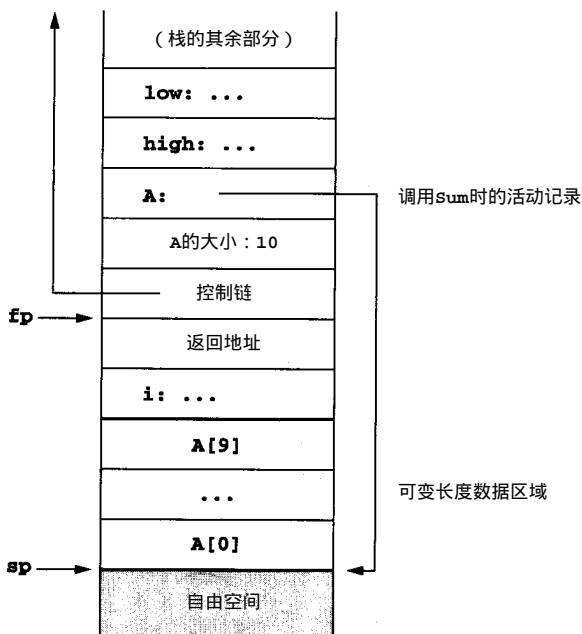
Ada 非约束数组 (unconstrained array) 是情况 (2) 的一个示例：

```
type Int_Vector is
    array(INTEGER range <>) of INTEGER;

procedure Sum (low, high: INTEGER;
               A: Int_Vector) return INTEGER
is
    temp: Int_Array (low..high);
begin
    ...
end Sum;
```

(请注意局部变量 **temp**，其大小不可预测)。处理这种情况的典型办法是：为变量长度数据使用间接的额外层，并将指针存放到一个在编译时可预测的地址中的实际数据里，同时执行期间用 sp 可管理的方法在运行栈的顶部进行真正的分配。

例 7.6 给定前面定义的 Ada **Sum** 过程，假定环境的组织也同上面一样[⊖]，那么可以如下所示实现 **Sum** 的活动记录（这个图示具体地显示了一个当数组大小为 10 时对 **sum** 的调用）：



现在，对 **A[i]** 的访问就可由计算

$@6 (fp) + 2 * i$

得到。其中 @ 意味着间接，且此时仍假设整型两个字节，地址 4 个字节。

注意，在上例所描述的实现中，调用程序必须知道 **sum** 的任何活动记录的大小。而且编译

⊖ 这在 Ada 中实际是不够的，它将会导致嵌套过程。参见本节后面的讨论。

程序也了解在调用点上的参数部分和簿记部分的大小（这是因为可以计算出自变量大小，而簿记部分与所有的过程都相同），但是一般而言，却不知道调用点上的局部变量部分的大小。因此，这个实现就要求编译程序为每个过程预先计算出局部变量的大小并将其存放在符号表中以备后用。用类似的方法可处理可变长度局部变量。

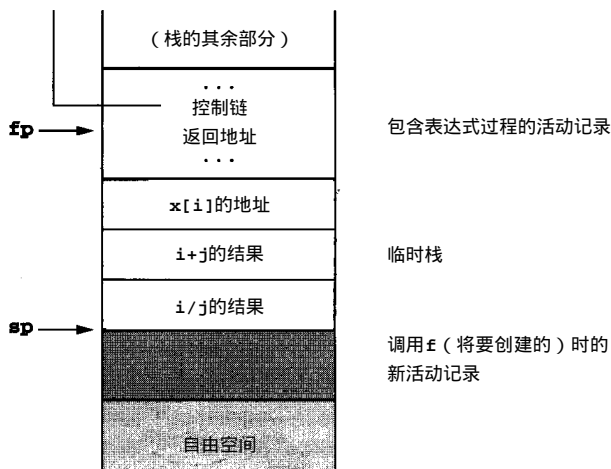
我们还需提醒大家C数组并不属于这类可变长度数据。实际上，C数组是一些指针，所以数组参数是由引用在C中传递且并不局部分配（而且它们不带有任何尺寸信息）。

4) 局部临时变量和嵌套声明 基于栈的运行时环境还有两个需要提及的复杂问题：局部临时变量和嵌套声明。

过程调用时必须保存的计算是导致局部临时变量的部分原因。例如考虑C表达式：

$$x[i] = (i + j) * (i/k + f(j))$$

在这个表达式从左到右的求值计算中，在对 f 的调用过程中需要保存中间结果： $x[i]$ 的地址（未决赋值）、计算 $i+j$ 的和（加法的未决）以及 i/k 的商（和与 $f(j)$ 的调用结果未决）。这些中间值可计算到寄存器中，并根据某个寄存器管理机制进行保存和恢复，或者可将它们作为临时变量存储在对 f 调用之前的运行时栈中。在这后一种情况下，运行时栈可能会出现在对 f 的调用之前的点上，如下所示：



在这种情况下，先前描述的利用 sp 的调用序列并未改变。此外，编译程序还可以很便利地从 fp 计算出栈顶的位置（在缺少变量长度数据时），这是因为临时变量所要求的数量是编译时决定的量。

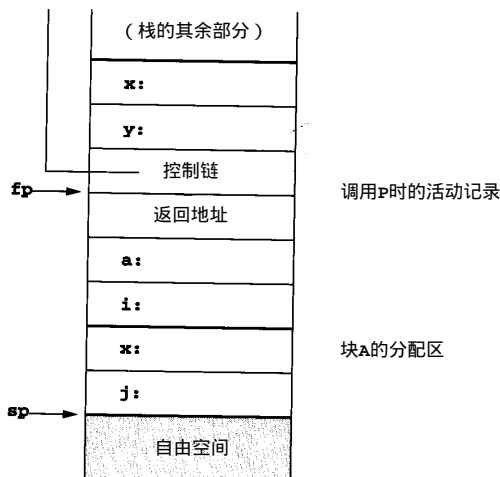
嵌套声明也出现了类似的问题。考虑以下的C代码

```
void p( int x, double y)
{ char a;
  int i;
  ...
  A:{ double x;
    int j;
    ...
  }
  ...
  B:{ char * a;
```

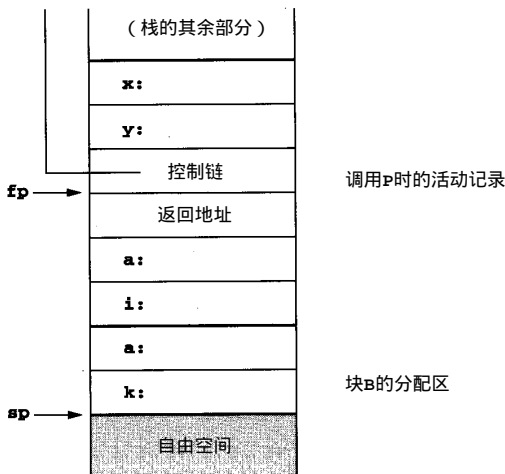
```
int k;  
...  
}  
...  
}
```

在这个代码中，在过程 *p* 的主体中嵌套着两个分别标作 *A* 和 *B* 的块（也称作复合语句），它们每个都有两个作用域仅仅覆盖着其所在块（也就是说向上直到下一个闭合的括号）的局部声明。在块进入之前无需对这些块的局部声明进行分配，而且块 *A* 和块 *B* 也无需同时进行分配。编译程序能够像对待过程一样处理块，并且在每次进入块时创建新的活动记录，并在退出时抛弃它。然而，由于这样的块比过程简单得多，所以它的效率并不高：这样的块没有参数且无返回地址，而且总是立即被执行而不是从其他地方调用。一个更简单的方法是按照与临时表达式相类似的办法在嵌套的块中处理声明，并在进入块时在栈中分配它们而在退出时重新分配。

例如，在上面所给出的简单 C 代码中进入块 *A* 之后，运行时栈应如下所示：



当进入块 *B* 后，则如下所示：



这样的实现必须这样小心地分配嵌套声明，周围过程块的 *fp* 的偏移在编译时计算。特别是，这

样的数据必须要在任何变量长度数据之前进行分配。例如在上面才给出的代码中，位于块 **A** 上的变量 **j** 从 **p** 的 **fp** 的偏移是 -17（再次假设整型 2 个字节，地址 4 个字节，浮点实数 8 个字节，而字符是 1 个字节），块 **B** 中的 **k** 的偏移是 -13。

7.3.2 带有局部过程的基于栈的环境

如果在语言编译时允许有局部过程声明，那么前面所讲到的运行时环境就无效了，因为没有提供非局部的和非全局的引用。

例如，考虑程序清单 7-4 中的 Pascal 代码（在 Ada 中也可以写出类似的程序）：在对 **q** 的调用中，运行时环境如图 7-5 所示。当使用标准的静态作用域规则时，在 **q** 中每次提及 **n** 必须指的是 **p** 的局部整型变量 **n**。正如我们在图 7-5 中所看到的一样，使用至今为止保存在运行时环境中的任何簿记信息都无法找到这个 **n**。

程序清单 7-4 Pascal 程序显示非局部的非全局的引用

```
program nonLocalRef;

procedure p;
var n: integer;

    procedure q;
    begin
        (* a reference to n is now
           non-local non-global *)
    end; (* q *)

    procedure r(n: integer);
    begin
        q;
    end; (* r *)

begin (* p *)
    n := 1;
    r(2);
end; (* p *)

begin (* main *)
    p;
end.
```

若我们愿意接受动态作用域，那么利用控制链有可能找到 **n**。观察图 7-5，看到通过跟随控制链就可以找到在 **r** 的活动记录中的这个 **n**，而且若 **r** 没有 **n** 的说明，则可以通过跟随另一个控制链来找到 **p** 的 **n**（这个处理称为链接（chaining），我们很快还会看到这个方法）。不幸的是，不仅仅是这个实现是动态作用域，可以找到 **n** 的偏移也会随着调用的不同而不同（请注意，在 **r** 中的 **n** 与在 **p** 中的 **n** 具有不同的偏移）。因此，在这样的实现中，必须在执行时保存用于每个过程的局部符号表，这样才能允许在每个活动记录中查询标识符，以及若它退出的话也可以看到，并且可以判定出它的偏移。这是运行时环境的最主要的额外复杂性。

解决这个问题的方法也实现静态作用域，是将一个称作访问链（access link）的额外簿记信息添加到每个活动记录中。除了可以指向代表过程的定义环境而不是调用环境之外，访

问链与控制链相似。正是由于这个原因，即使它不是编译时决定的量，访问链有时也被称作静态链 (static link) ^①。

图7-6显示了将图7-5中的运行时栈修改之后包括了访问链的情况。在这个新的环境之下，*r*和*q*的活动记录的访问链都指向*p*的活动记录，这是*r*和*q*都在*p*中声明的缘故。因为这总是*p*的一个活动记录，现在位于*q*中的对*n*的引用会引起后接访问链，这里*n*可在固定偏移处找到。通常，通过将访问链装载到寄存器中，然后根据到这个寄存器（它此时作为*fp*）的偏移访问*n*，而在代码中完成。例如，使用前面描述的大小约定，若寄存器*r*用作访问链，则在用值4(*fp*)装载*r*之后可将*p*中的*n*作为-6(*r*)来访问（访问链从图7-6中的*fp*得到偏移+4）。

注意，正如由它将要到达的位置上的括号中的注解所指出的，过程*p*的活动记录本身并未包含访问链。这是因为*p*是一个全局过程，所以*p*中的任何非局部的引用必须都是全局引用且通过全局引用机制来访问。因此，访问链就是多余的了（实际上为了与其他的进程保持连贯性，可以很便利地插入空的或是任意的访问链）。

上面讨论的情况实际是最简单的，其中非局部引用是指向下一个最外面的作用域中的声明。而指向最远的作用域中的声明的非局部引用也是可能的。例如在程序清单7-5中的代码。

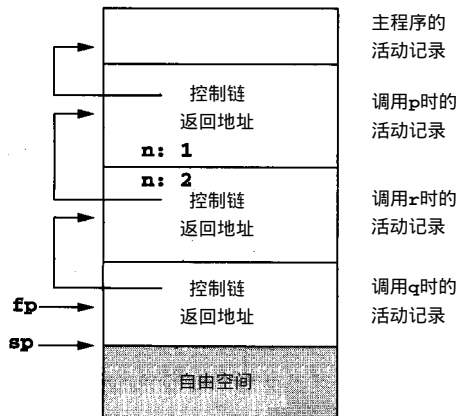


图7-5 程序清单7-4中程序的运行时栈

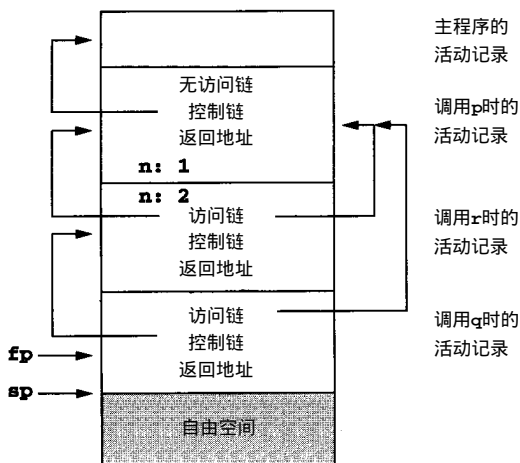


图7-6 添加了访问链后的程序清单7-4中程序的运行时栈

程序清单7-5 示范访问链的Pascal代码

```

program chain;

procedure p;
var x: integer;

  procedure q;
  procedure r;
  begin
    x := 2;
    ...
    if ... then p;
  end; (* r *)

```

① 我们当然了解定义过程，但却不知道它的活动记录的确切位置。


```

begin
  r;
end; (* q *)

begin
  q;
end; (* p *)

begin (* main *)
  p;
end.

```

在这段代码中，在过程 q 中说明了过程 r ，而过程 q 又是在过程 p 中说明的。因此，对于 r 中的 x 的赋值（即 p 的 x ）必须越过两个作用域层去寻找 x 。图7-7显示了对 r 的（第1个）调用之后的运行时栈（由于 r 可能递归地调用 p ，所以到 r 的调用可能不止一个）。在这种环境中，必须跟随两个访问链才能到达 x ，这个过程称作访问链接（access chaining）。访问链接是通过重复地取出访问链实现的，利用前面取出的链好像它就是 fp 。图7-7中的 x 可如下进行访问（使用前面的大小约定）：

Load 4(fp) into register r .

Load 4(r) into register r .

Now access x as $-6(r)$.

对于用于访问链工作的方法，编译程序必须能够在局部访问名字之前判定出要链接多少个嵌套层。这就要求编译程序预先为每个声明计算出嵌套层（nesting level）属性。通常，将最远的作用域（Pascal中的主程序层或是C中的外部作用域）给定为嵌套层0，每次进入一个函数或过程（在编译时），嵌套层就增加1，退出时则减去1。例如，在程序清单7-5的代码中，由于过程 p 是全局的，所以它的嵌套层为0；由于在进入 p 时嵌套层增加了，所以变量 x 的嵌套层为1；由于过程 q 对于 p 是局部的，所以它的嵌套层也为1；而过程 r 的嵌套层为2的原因是当进入 q 时嵌套层再次增加了。最后，在 r 内的嵌套再一次增加到3。

现在通过比较在访问点上的嵌套层与名

字声明的嵌套层，可判断出访问非局部名字所必须的链接的数量；后面跟随的访问链接数是这两个嵌套层的差。例如，在前面的情况中，对 x 的赋值发生在嵌套层3，而 x 具有嵌套层1，所以必须跟在两个访问链接之后。一般而言，若在嵌套层中的差是 m ，那么为访问链接而必须生成的代码须将 m 装入到一个寄存器 r 上，且使第1个使用 fp ，其他的用 r 。

由于必须为每个带有大的嵌套差的非局部引用执行一个很长的指令序列，所以对于变量访问访问链接看起来效率不高。但在实际运用中，嵌套层极少有超过两个到3个深度的，而且大多数的非局部引用都是对全局变量的（嵌套层为0），这样就可利用前面所讲到的直接办法对它们继续访问了。在嵌套层索引的查询表中有一个实现访问链接方法，链接不会带来执行开销。该方法中所用到的数据结构称作显示（display）。它的结构与使用在练习中论述。

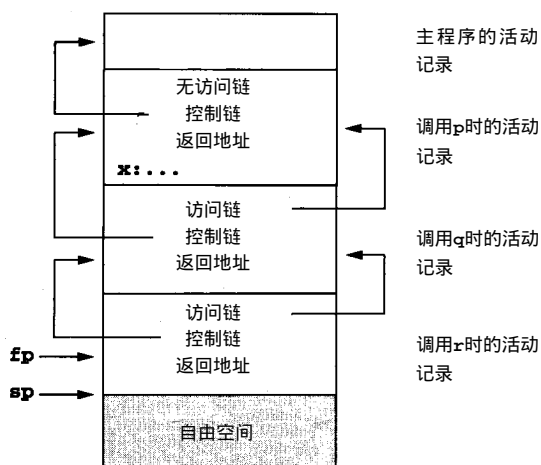


图7-7 程序清单7-5代码中第1次对 r 的调用之后的运行时栈

1) 调用序列 实现访问链接的调用序列的改变相对比较简单。在实现中, 调用时必须将访问链压入到 fp 之前的运行时栈中, 退出之后必须用一个额外的量来修改 sp 以便像对自变量一样删掉访问链接。

唯一的问题是在调用时寻找过程的访问链接。通过使用附在正调用的过程声明之上的 (编译时) 嵌套层信息可以解决这个问题。实际上, 我们所要做的仅是生成一个访问链接, 就像在过程调用的同一嵌套层上访问一个变量。这样计算所得出的地址就是相应的访问链。当然若过程是局部的 (在嵌套层中的差是 0), 那么访问链与控制链就是相同的 (而且也与在调用点上的 fp 相同)。

例如可考虑程序清单 7-4 上的 r 中对 q 的调用。在 r 内, 位于嵌套层 2, 而 q 的声明在嵌套层 1 中 (这是因为 q 对于 p 而言是局部的, 而在 p 内的嵌套层是 1)。因此, 一个访问步骤就要求计算 q 的访问链, 而在图 7-6 中, q 的访问链指向 p 的活动记录 (且与 r 的访问链相同)。

请注意, 即使是位于定义环境的多重活动中, 这个过程也将计算出正确的访问链, 因为计算是在运行时而不是在编译时进行的 (利用编译时嵌套层)。例如, 假设有程序清单 7-5 的代码, 在对 r 的第 2 个调用后 (假定是对 p 的递归调用), 运行时栈如图 7-8 所示。在该图中, r 有两个不同的活动记录, 带有两个不同的访问链, 指向 q 的不同的活动记录, 代表 r 不同的定义环境。

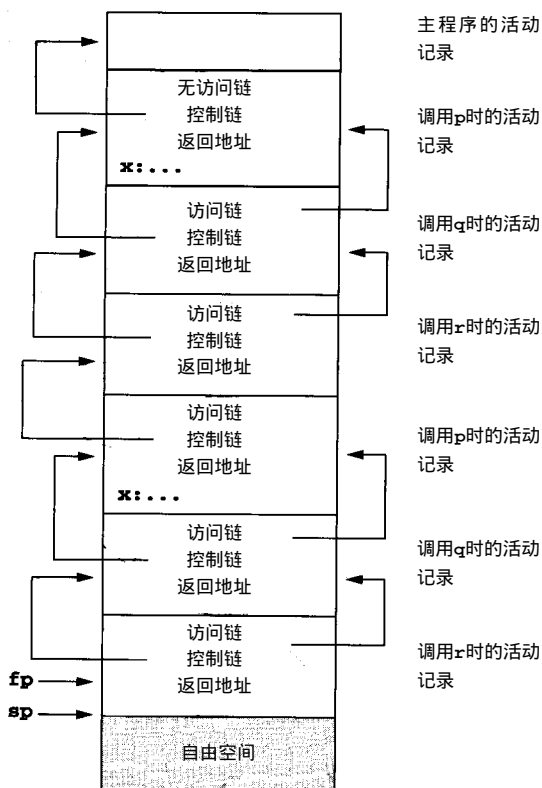


图 7-8 在程序清单 7-5 的代码中对 r 的第 2 次调用之后的运行时栈

7.3.3 带有过程参数的基于栈的环境

在某些语言中, 不仅允许有局部过程, 而且还可将过程作为参数传递。在这样的语言中, 当调用一个作为参数传递的过程时, 编译程序不可能像前一节所讲的那样生成代码以计算调用点上的访问链。当将过程作为参数传递时, 必须预先计算出过程的访问链并与过程代码的指针一同传递。因此, 再也不能将过程参数值看作是一个简单的代码指针了, 它应包含一个访问指针, 定义解决非局部引用的环境。这个指针对, 一个代码指针和一个访问链, 或一个指令指针 (instruction pointer) 和环境指针 (environment pointer), 一同表示了过程或函数参数的值, 它们通称为闭包 (closure) (这是因为访问链“闭合了”由非局部引用引起的“洞”)。我们将闭包表示为 $\langle ip, ep \rangle$, 其中 ip 表示过程的指令指针 (代码指针或入口点), 而 ep 表示过程的环境指针 (访问链)。

⊖ 这个术语在微积分中有它自己的来源, 且它不会与正则表达式或 NFA 状态中的 ϵ 闭包的 (Kleene) 闭包运算相混淆。

例7.7 考虑程序清单7-6中的标准Pascal程序，它有一个过程 p ，带有一个也是过程的参数 a 。在 q 中对 p 调用之后， q 的过程 r 传递到 p ， p 中的对 a 的调用实际上调用的是 r ，而且这个调用仍必须在 q 的活动中寻找非局部变量 x 。当调用 p 时，将 a 构造为闭包 $\langle ip, ep \rangle$ ，其中 ip 是指向 r 的代码的指针，而 ep 是在调用点 fp 的拷贝（也就是它指向调用 q 的环境，其中定义了 r ）。 a 的 ep 的值由图7-9的虚线指明，表示在 q 中的调用 p 之后的环境。接着当在 p 内调用 a 时，就将 a 的 ep 用作其活动记录的静态链，如图7-10所示。

程序清单7-6 带有作为参数的过程的标准Pascal代码

```

program closureEx(output);

procedure p(procedure a);
begin
  a;
end;

procedure q;
var x:integer;

  procedure r;
  begin
    writeln(x);
  end;

begin
  x := 2;
  p(r);
end; (* q *)

begin (* main *)
  q;
end.

```

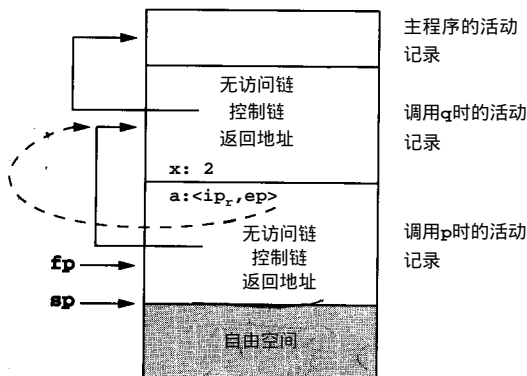


图7-9 程序清单7-6的代码中调用 p 之后的运行时栈

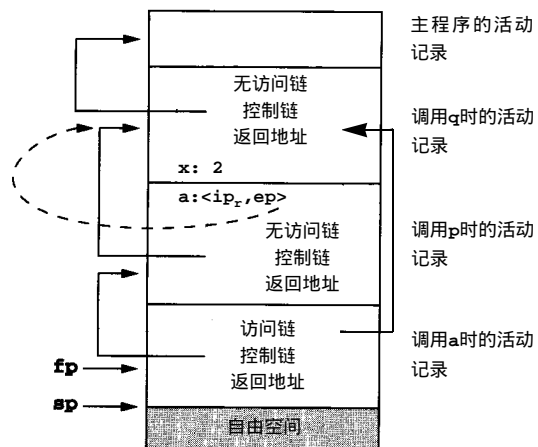


图7-10 程序清单7-6的代码中调用 a 之后的运行时栈

现在，刚刚描述的环境中的调用序列能很清楚地区分常规过程和过程参数。同前面一样，

常规过程调用是使用过程的嵌套层取出访问链，并直接跳到过程的代码（在编译时已知）。而另一方面，过程参数早已得到了它的访问链，并已将其存放在局部活动记录中，而这一记录又必须取出并插入到新的活动记录中。然而，编译程序却无法直接得到过程代码的地址；而必须对存放在当前活动记录中的 ip 进行间接调用。

为了保持简洁和一致性，编译程序的编写者可能会希望要避免常规过程与过程参数之间的这种区别，并将所有的过程都作为环境中的闭包。实际上，若语言对过程的处理越普通，则这个方法就越合理。例如，若允许有过程变量，或可以动态地计算过程变量，则过程的 $\langle ip, ep \rangle$ 表示就变成了对这种情形的要求了。图 7-11 显示了当所有的过程值都存放在作为闭包的环境中时图 7-10 的环境。

最后，我们注意到 C、Modula-2 和 Ada 都避免本节所提到的复杂情况：对于 C 而言，它没有局部过程（即使它有参数和变量）；对于 Modula-2 而言，是一个特殊的规则限制了过程参数和过程变量值都应是全局过程；而对于 Ada 而言，则是由于它没有过程参数和变量。

7.4 动态存储器

7.4.1 完全动态运行时环境

上一节讨论的基于运行时环境的栈在 C、Pascal 以及 Ada 这样的标准命令式语言中是最普通的环境格式。但这样的环境也有限制。尤其是如果一种语言在过程中对局部变量的引用可返回到调用程序，无论是显式的还是隐含的，在过程退出时的基于栈的环境都会导致摇摆引用（dangling reference），这是因为过程的活动记录将从栈中释放分配。最简单的示例是返回局部变量的地址，如在 C 代码中：

```
int * dangle(void)
{ int x ;
  return &x; }
```

现在赋值 $addr = dangle()$ 使 $addr$ 指向活动栈中的不安全的地址，它的值可由后面对任何过程的调用随机改变。C 对此类问题的处理是，只说明这样的程序是错误的（尽管没有哪个编译程序会给出错误信息）。换言之，C 的语义被建立在基于栈的环境之下。

若调用可返回局部函数，则会发生更为复杂的摇摆引用情况。例如，如 C 允许有局部函数定义，则程序清单 7-7 的代码就会出现一个对 x 和 g 的参数的间接摇摆引用，在 g 退出后调用 f 就可访问到它了。当然 C 是通过禁止局部过程的存在来避免这个问题的。诸如 Modula-2 的其他语言既有局部过程也有过程变量、参数和返回值，就必须定出一个特殊规则使程序报错（在 Modula-2 中，该规则是仅有全局过程才可以是自变量或返回值——这甚至是从 Pascal 风格过程

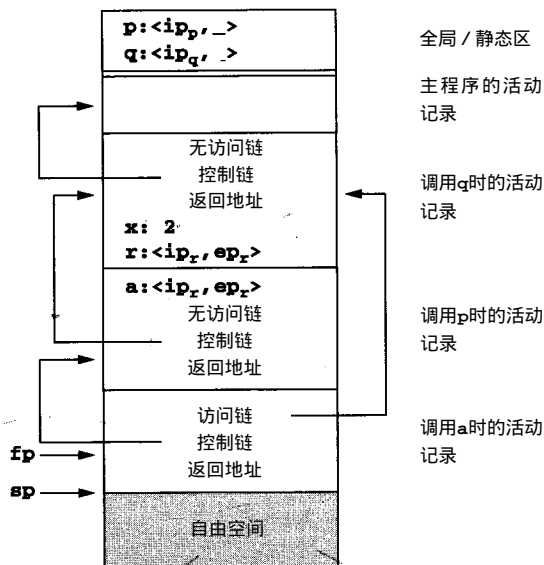


图 7-11 程序清单 7-6 的代码调用 a 之后的运行时栈，所有的过程都作为环境中的闭包

变量的主要退步)。

程序清单 7-7 显示返回局部函数引起的摇摆引用的伪 C 代码

```
typedef int (* proc)(void);

proc g(int x)
{ int f(void) /* illegal local function */
  { return x;}
  return f; }

main()
{ proc c;
  c = g(2);
  printf("%d\n",c()); /* should print 2 */
  return 0;
}
```

但是在很多语言中，这样的规则并不适用，即那些像 LISP 和 ML 的函数程序设计语言。设计函数语言的一个主要原则是函数应尽可能的通用，而这就意味着函数应是能够局部定义的，并像参数一样传递，作为结果返回。因此，对于此类的语言而言，基于栈的运行时环境并不合适，而且需要一个更一般的环境格式。因为活动记录仅在对它们所有的引用都消失了才再重新分配，而且这又要求活动记录在执行时可动态地释放在任意次，所以称这个环境为完全动态的 (fully dynamic)。因为完全动态运行时环境包含了要在运行时跟踪引用，以及在执行时任意次地找寻和重新分配存储器的不可访问区域 (这种处理称作废弃单元收集 (garbage collection))，所以这种环境比基于栈的环境要复杂许多。

尽管在这个环境中增加了的复杂性，其活动记录的基本结构仍保持不变：必须为参数和局部变量分配空间，而且仍然需要控制链和访问链。当然，现在当控制返回到调用程序时（且使用控制链来恢复先前的环境），退出的活动记录仍留在存储器中，而且在以后的某个时刻被重新分配。因此这个环境的整个额外的复杂性可被压缩到存储器管理程序中，这个管理程序将取代带有更普通的分配和重新分配例程的运行时栈操作。本节后面还会谈到一些有关这样的存储器管理程序的设计问题。

7.4.2 面向对象的语言中的动态存储器

面向对象的语言在运行时环境中要求特殊的机制以完成其增添的特征：对象、方法、继承以及动态装订。这一小节将给出有关这些特征的各种实现技术。我们假设读者对于基本的面向对象的技术和概念比较熟悉^①。

面向对象语言在对运行时环境方面的要求差异很大。Smalltalk 和 C++ 是这种差异的极好的代表者：Smalltalk 要求与 LISP 相似的完全动态环境；而 C++ 则在设计上花了很大的功夫以保持 C 的基于栈的环境，它并不需要自动动态存储器管理。在这两种语言中，存储器中的对象可被看作是传统记录结构和活动记录之间的交叉，且带有作为记录域的实例变量（数据成员）。这个结构与传统记录在对方法和继承特征的访问上有着一定的差别。

实现对象的一个简单机制是，初始化代码将所有当前的继承特征（和方法）直接地复制到记

^① 以后的讨论也假设只有继承性是可用的。“注意与参考”部分中的某些著作还谈到了多重继承性。

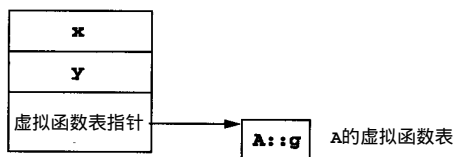
录结构中(将方法当作代码指针)。但这样做极浪费空间。另外一种方法是在执行时将类结构的一个完整的描述保存在每个点的存储器中,并由超类指针维护继承性(有时这也称作继承图(inheritance graph))。接着同用于它的实例变量的域一起,每个对象保持一个指向其定义类的指针,通过这个类就可找到所有(局部和继承的)的方法。此时,只记录一次方法指针(在类结构中),而且对于每个对象并不将其复制到存储器中。由于是通过类继承的搜索来找到这个机制的,所以该机制还实现继承性与动态联编。其缺点在于:虽然实例变量具有可预测的偏移量(正如在标准环境中的局部变量一样),方法却没有,而且它们必须由带有查询功能的符号表结构中的名字维护。然而,它是对于诸如 Smalltalk 的高度动态语言的合理的结构,其中对于类结构的改变可以发生在执行中。

将整个类结构保存在环境中的另一种方法是,计算出每个类的可用方法的代码指针列表,并将其作为一个虚拟函数表(virtual function table)(C++术语)而存放在(静态)存储器。它的优点在于:可做出安排以使每个方法都有一个可预测的偏移量,而且也就不再需要用一系列表查询遍历类的层次结构。现在每个对象都包括了一个指向相应的虚拟函数表而不是类结构的指针(当然,这个指针的位置必须也有可预测的偏移量)。这种简化仅在类结构本身是固定在执行之前的情况下才成立。它是 C++ 中选择的方法。

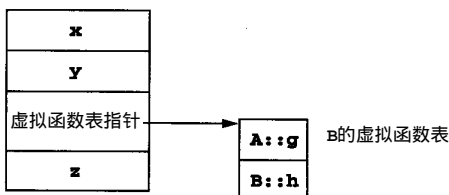
例7.8 考虑以下的C++类声明:

```
class A
{ public:
  double x,y;
  void f();
  virtual void g();
};
class B:public A
{ public:
  double z;
  void f();
  virtual void h();
};
```

类A的一个对象应出现在存储器中(带有它的虚拟函数表),如下所示:



而类B的一个对象则应如下所示:



每次增添对象结构时,注意虚拟函数指针如何保留固定的地址,这样就可执行之前知道

它的偏移量。还应注意(由于函数f没有声明“虚拟”),它并不遵守C++动态联编,因此也就不出现在虚拟函数表(或环境中的任何其他地方):在编译时决定对f的调用。

7.4.3 堆管理

在7.4.1节中,我们讨论了如果完全支持一般的函数,那么与在大多数编译语言使用的基于栈的运行时环境比,更需要具有动态性。但在绝大多数语言中,即使是基于栈的环境也需要一些动态功能以处理指针分配和重新分配。处理这样的分配的数据结构称作堆,堆通常作为存储器中的一个线性块分配,这样如果需要它还可以生长,而且对栈的干扰尽可能小(7.1节已显示了堆位于栈区域的相反一端的存储器块中)。

本章一直到这里,我们的注意力都是放在活动记录和运行时栈的组织上。而在本节中,我们希望描述一下如何管理堆,以及如何将堆操作扩展,提供带一般函数功能的语言要求的动态分配。

堆提供两个操作:分配操作和释放操作。分配操作通常是按字节数得到一个大小参数(或显式或隐含),并返回一个指向正确大小的存储器块的指针,或若不存在则返回一个空指针。释放操作得到一个指向被分配的存储器块的指针并再次将它标为空的(释放操作必须还能通过或显式或隐含的参数来发现将空的块的大小。)这两个操作存在于许多语言的不同名称之下:在Pascal分别称作new和dispose,在C++中称作new和delete。C语言中有这些操作的若干个版本,但最基本的是malloc和free,它们都是标准库(stdlib.h)的一部分,此时它们基本都有以下的声明:

```
void * malloc (unsigned nbytes);
void free (void * ap);
```

我们将用这些声明作为堆管理的基本描述。

维护堆和实现这些函数的标准方法是使用空块的环形链接列表, malloc从中得到存储器而free返回存储器。它具有简单的优点,但也有缺点:其一, free操作不能辨认出它的指针自变量是否是它真正指向的由 malloc先前分配的合法块。若用户传递了一个无效的指针时,则堆就会很容易和很快坏掉。其二(问题并不很严重)是必须注意,返回处附近有空的块列表时要合并(coalesce)块,因为这样会导致最大的空块。若不合并,堆就会很快变成碎片(fragmented),也就是被分割成大量的较小的块,这样尽管有足够多的全部可用空间可用于分配,但分配大块时却失败了(在合并中当然也可能有碎片)。

在这里我们指出使用环形链接的列表数据结构的 malloc和free在实现上的一点差异,这个数据结构保存分配的和空的块(因此也就不易坏掉),也提供在自合并块方面的优点。程序清单7-8给出了代码。

程序清单 7-8 维护邻近的存储器的C代码,使用指向使用块和空块的指针

```
#define NULL 0
#define MEMSIZE 8096 /* change for different sizes */

typedef double Align;
typedef union header
{ struct { union header *next;
          unsigned usedsize;
          unsigned freesize;
        } s;
  Align a;
} Header;
```



```

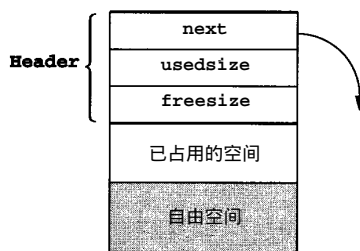
static Header mem[MEMSIZE];
static Header *memptr = NULL;

void *malloc(unsigned nbytes)
{ Header *p, *newp;
  unsigned nunits;
  nunits = (nbytes+sizeof(Header)-1)/sizeof(Header) + 1;
  if (memptr == NULL)
  { memptr->s.next = memptr = mem;
    memptr->s.usedsize = 1;
    memptr->s.freesize = MEMSIZE-1;
  }
  for(p=memptr;
      (p->s.next!=memptr) && (p->s.freesize<nunits);
      p=p->s.next);
  if (p->s.freesize < nunits) return NULL;
  /* no block big enough */
  newp = p+p->s.usedsize;
  newp->s.usedsize = nunits;
  newp->s.freesize = p->s.freesize - nunits;
  newp->s.next = p->s.next;
  p->s.freesize = 0;
  p->s.next = newp;
  memptr = newp;
  return (void *) (newp+1);
}

void free(void *ap)
{ Header *bp, *p, *prev;
  bp = (Header *) ap - 1;
  for (prev=memptr, p=memptr->s.next;
      (p!=bp) && (p!=memptr); prev=p, p=p->s.next);
  if (p!=bp) return;
  /* corrupted list, do nothing */
  prev->s.freesize += p->s.usedsize + p->s.freesize;
  prev->s.next = p->s.next;
  memptr = prev;
}

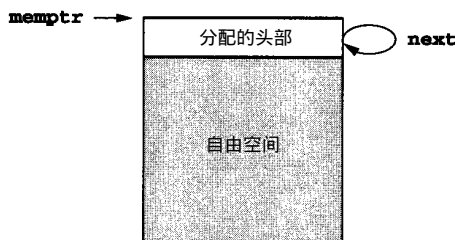
```

这个代码使用容量 MEMSIZE 的静态分配数组作为堆，但也可使用操作系统调用分配堆。我们定义了一个数据类型 Header 保存每个存储器块的簿记信息，定义了具有 Header 类型元素的堆数组，这样就可很容易地将簿记信息保存在存储器块中。类型 Header 包含了 3 块信息：指向列表的下一个块的指针，当前分配空间的长度（位于存储器之后），以及任何后面的自由间的长度（若有的话）。因此，列表中的每个块都有格式

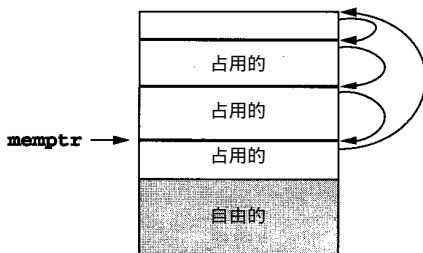


程序清单 7-8 中类型 `Header` 的定义还使用了一个 `union` 声明和 `Align` 数据类型 (在代码中将其设为 `double`)。这是将存储器元素排在合理的字节边界上, 根据系统的不同, 这有时是需要的, 有时是不需要的。后面的描述中可安全地把这种复杂性忽略掉。

堆操作还需要的另一片数据是指向环形链接的列表中的一个块的指针。这个指针称作 `memptr`, 它总是指向具有一些自由空间的块 (通常是被分配或释放的最后一个空间)。它被初始化为 `NULL`, 但是在 `malloc` 的第一次调用上, 对初始化代码的执行是通过将 `memptr` 设置为堆数组的开头并初始化数组的头部, 如下所示:



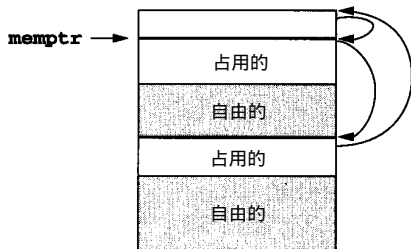
这个在第 1 次调用 `malloc` 时分配的初始化头部永远也不会被释放。这时在列表中有一个块, 而其余的 `malloc` 代码搜索该列表并从具有足够自由空间的第 1 个块中返回一个新块 (这是首次适用 (first fit) 算法)。因此在对 `malloc` 的 3 次调用之后, 该列表看起来应是这样的:



注意, 当连续分配块时, 每次都会生成一个新块, 并且还有前面块所剩下的自由空间 (因此从块的自由空间生成的分配总是将 `freysize` 设置为 0)。 `memptr` 跟随在新块的构造之后, 所以它总是指向某个自由空间的块。大家还要注意, `malloc` 总是增加指向新创建块的指针, 所以也将头部保护起来而不会被客户程序覆盖 (只要在返回存储器中使用正向偏移)。

现在来考虑 `free` 过程的代码。它首先把用户传递的指针减 1, 以找到块的头部。接着它再搜索列表以寻找与之相同的指针, 以保护该列表防止坏掉, 而且还能计算指向先前块的指针。一旦找到就将该块从列表中删除, 且将其使用过的和自由空间都添加到先前块的自由空间中, 所以也就自动地合并了自由空间。请读者注意, 还将 `memptr` 设置为指向包含了刚才释放的存储器的块。

例如, 假设将上图中 3 个使用过的块的中间一个释放了, 则堆和与之相关的块列表应如下所示:



7.4.4 堆的自动管理

由于程序员必须编写出到分配和释放存储器的明确的调用, 所以用 `malloc` 和 `free` 完成指针的动态分配和重新分配是管理堆的手工 (manual) 方法。相反地, 运行时栈则是由调用序列自动地 (automatically) 管理。在一种需要完全动态的运行环境语言中, 堆也必须类似地自动管理。然而尽管在每个过程调用中可以很方便地调度对 `malloc` 的调用, 但是由于活动记录必须要持续到其所有的引用都消失为止, 所以退出时却很难调度对 `free` 的调用。因此, 自动存储器管理涉及到了前面分配的但不再使用的存储器的回收, 可能是在它被分配的很久以后, 而没有明确的对 `free` 的调用。这个过程称作垃圾回收 (garbage collection)。

在存储块不再引用时, 无论是直接还是通过指针间接的引用, 识别是一项比维护堆存储块的列表复杂得多的任务。标准的技术是执行标识和打扫 (mark and sweep) 垃圾回收^①。在这种方法中, 直到一个对 `malloc` 的调用失败之前都不会释放存储器, 在这时将垃圾回收程序激活, 寻找可被引用的所有存储器并释放所有未引用的存储器。这是通过两遍来完成的。第 1 遍递归地顺着所有的指针前进, 从所有当前的可访问指针值开始, 并标出到达的每个存储块。这个过程要求额外的位存储标识。另一个遍则线性地打扫存储器, 并将未标出的块返回到自由存储器中。虽然这个过程通常要寻找足够的相邻自由存储器以满足一系列的新要求, 但存储器仍有可能是非常破碎, 故尽管是在垃圾回收之后, 大的存储请求仍旧会失败。因此, 垃圾回收经常也会通过将所有的分配的空间移到堆的末尾, 以及在另一端留下相邻的自由空间的唯一一个大型块而执行存储器压缩 (memory compaction)。这个过程还必须在存储器中更新对那些在执行程序时被移掉的区域的所有引用。

标识和打扫垃圾回收有若干个缺点: 它要求额外的存储 (用于标识), 在存储器中的两个遍导致了过程中很大的延迟, 有时需要几秒钟, 而每一次调用垃圾回收程序又都需要几分钟时间。这对于那些许多涉及到了交互和即时响应的应用程序显然是不合适的。

可以通过将可用的存储器分为两个部分并每次只从一个部分中分配存储来对这个过程进行改进。在标识遍时, 将所有到达了的块都复制到未被使用的另一半存储器中。这就意味着在存储时不再要求额外的标识位而且一个遍就够了。它还自动地进行压缩。一旦位于使用的区域中的所有可到达的块都复制好时, 就将使用的和未使用的存储器部分相互交换, 而过程依然继续进行。这种方法称作停机和复制 (stop-and-copy) 或二部空间 (two space) 垃圾回收。然而它对存储回收中的过程延迟改进不大。

最近又提出了一个大大减少延迟的方法, 称为生育的垃圾回收 (generational garbage collection), 它将一个永久的存储区域添加到前一段描述的回收方案中。将存在时间足够长的被分配的对象只复制到永久空间中, 并在随后的存储回收时不再重新分配。这就意味着垃圾回收程序在更新的存储分配时只需要搜索存储器中的很小的一个部分。当然永久存储器也有可能由于不可达到的存储而用尽, 但这相对于前面的问题就不那么严重了, 这是因为临时存储会很快消失, 而可被分配的存储则总会有的。人们已证明了这个处理很好, 在虚拟存储系统中尤为如此。

读者可在“注意与参考”一节中查阅此种方法的具体细节以及其他的垃圾回收方法。

7.5 参数传递机制

我们已经看到了在过程调用中, 参数是如何通过调用程序在跳到被调用过程的代码之前与

① 另一种称为引用计数 (reference counting) 的更为简单的方法也经常用到。参见“注意与参考”一节。

活动记录中的位置相对应的,该活动记录则由自变量或参数值组成。因此对于被调用的过程而言,参数代表了没有附加代码的完全正式的值,但该值仅在代码可以发现其最终值的活动记录中建立一个位置,这一最终值只在发生调用时才退出。建立这些值的过程有时是自变量的参数的绑定(binding)。自变量的值是如何由过程代码解释依赖于源语言的采用的特定参数传递机制(parameter passing mechanism(s))。正如早已提到过的,FORTRAN77采用的就是将参数传递到位置而不是值的机制,而C则将所有的自变量都当作是值。其他的语言,诸如C++、Pascal和Ada则是提供参数传递机制的选择。

在本节中,我们将讨论两个最常用的参数传递机制,值传递(pass by value)和引用传递(pass by reference)(有时也称作由值调用和由引用调用),此外还有两个重要方法,由值的结果传递(pass by value-result)和由名字传递(pass by name)(也称作延迟赋值(delayed evaluation))。它们的一些变形则放到了练习中。

未由参数传递机制本身说明的一个问题是自变量计算的顺序。在大多数情况下,这个顺序对于程序的执行并不重要,而且任何的计算顺序都是产生相同的结果。此时为了有较高的效率或其他原因,编译程序可能会选择改变自变量计算的顺序。但是许多语言却允许会导致副作用的自变量调用(改变存储器)。例如C的函数调用

```
f(++x,x);
```

会使x的值改变,所以不同的计算顺序会引起不同的结果。在这样的语言中,可能会要指定诸如从左到右的标准顺序,或者由编译程序的编写者来决定,而此时调用的结果在各种实现中都各不相同。特别地,C编译程序是从右到左计算它们的自变量。这就允许有不同数量的自变量(例如在printf函数中),如在7.3.1节中的讨论。

7.5.1 值传递

在这个机制中,自变量是在调用时计算的表达式,而且在执行过程时,它们的值就成为了参数的值。这是在C中唯一可用的参数传递机制,且在Pascal和Ada中是缺省的(Ada还允许将这样的参数显式地指定为传入(in)参数)。

在最简单的格式中,这就意味着值参数在执行过程中是作为常量值,而且可将值传递解释为用自变量的值取代过程体中的所有参数。Ada使用这个值传递的格式,此时不能给这样的参数赋值或作为局部变量来使用。C和Pascal采用更宽松的观点,在其中的值参数在本质上被看作是被初始化了的局部变量,该变量可被用作常规变量,但是它们的改变不会引起任何非局部的改变。

在诸如C这样仅提供值传递的语言中,通过改变它的参数直接写一个过程来达到目的是不可能的。例如,以下用C中写出的inc2函数并没有达到其预想的效果:

```
void inc2( int x)
/* incorrect! */
{ ++x; ++x; }
```

但在理论上,可能用函数恰当的一般性,通过返回相应的值而不是改变参数值来完成所有的计算,像C这样的语言通常提供使用值传递的方法,进行非局部改变。在C中,它使用了传递地址而不是值的格式(而且因此改变了参数的数据类型):

```
void inc2( int* x)
/* now ok */
{ ++(*x); ++(*x); }
```

当然由于要求y的地址而不是它的值,所以增加变量y,这个函数必须被称作inc2(&y)。

由于数组是隐含指针，这种方法在 C 中用于数组时特别好，而且值传递允许改变单个的数组元素：

```
void init(int x[],int size)
/* this works fine when called
   as init(a), where a is an array */
{ int i;
  for(i=0;i<size;++i) x[i]=0;
}
```

值传递在编译程序上没有特殊的要求。通过对自变量进行最直接的计算和活动记录的构造就可很便利地完成它了。

7.5.2 引用传递

在引用传递中，自变量必须与分配的地址一起变化（至少是原则上）。并非传递变量的值，引用传递的是变量的地址，因此参数就变成了自变量的别名（alias），而且在参数上发生的任何变化都会出现在自变量上。在 FORTRAN77 中，引用传递是唯一的参数传递机制的。在 Pascal 中，通过使用 var 关键字来得到引用传递，而在 C++ 中，则是通过在参数说明中使用特殊字符 &：

```
void inc2( int & x)
/* C++ reference parameter */
{ ++x; ++x; }
```

现在就可调用这个函数而无需特别使用地址操作符：inc2(y) 工作地很好。

引用传递要求编译程序计算自变量的地址（它也必须具有这样的地址），这个自变量将存放在局部活动记录中。由于局部“值”实际上是环境中的别处的地址，所以编译程序还要将对引用参数的局部访问转为间接访问。

在诸如 FORTRAN77 这样的只可用引用传递的语言中，通常要为是不带地址的值的自变量提供一个位置。像

```
p(2+3)
```

的调用在 FORTRAN77 中是非法的，编译程序必须为表达式 2+3 “创造” 出一个地址，并将值计算到这个地址中，接着再将地址传递到调用中。一般地，这通过在调用程序的活动记录中创建一个临时地址进行（在 FORTRAN77 中它是静态的）。在练习 7.1 中有这方面的一个例子，通过为它在主过程的活动记录中创建一个存储器地址将值 3 作为自变量传递。

引用传递的一个方面是它不要求复制被传递的值，这与值传递不同。当要复制的值是一个较大的结构（或是在除了 C 或 C++ 之外的语言中的一个数组）时，这有时会很重。在这种情况下，能够由引用来传递自变量就是很重要的了，但是它禁止自变量的值有任何变化，因此就能够做到值传递而无需覆盖值的拷贝。这个选项是由 C++ 所提供的，在其中可将调用写作：

```
void f( const MuchData & x )
```

其中的 MuchData 是带有大型结构的数据类型。这仍是引用传递，但是编译程序还必须执行一个静态检查：x 从不出现在一个赋值的左边或是被改变[⊖]。

⊖ 在完全安全的方法中并不能总是这样做。

7.5.3 值结果传递

除了未建立真正的别名之外，这个机制得到的结果与引用传递类似：在过程中复制和使用自变量的值，然后当过程退出时，再将参数的最终值复制回自变量的地址。因此，这个方法有时也被称为复制进，复制出，或复制存储。这是 Ada 的传入(in)传出(out)参数机制(Ada 还有一个简单的传出(out)参数，该参数没有传递进的初始值：这可称作结果传递)。

值结果传递与引用传递的唯一区别在于别名的表现不同。例如，在以下的代码中 (C 语法)：

```
void p(int x, int y)
{ ++x;
  ++y;
}

main()
{ int a=1;
  p(a,a);
  return 0;
}
```

在调用 p 之后，若使用了引用传递，则 a 的值为 3；若使用了值结果传递，则 a 的值为 2。

这个机制尚未指定的问题，可能随语言或实现的不同而不同，包括复制到自变量的结果的顺序以及自变量的地址是否仅在入口处计算和存储，或是否在退出时重新计算。

Ada 还有一个问题：它的定义说明传入 (in) 传出 (out) 参数会通过引用作为遍来真正地实现，而且任何在两个机制之下不同的计算 (由此就涉及到了别名) 都是错误的。

从编译程序的编写者的观点来看，值结果传递要求对运行时栈以及调用序列的基本结构的进行若干个修改。首先，被调用的程序不能释放活动记录，这是因为复制出的 (局部) 值还必须对调用程序适用。其次，调用程序必须或是在建立新的活动记录开始之前就将自变量的地址压入到栈中，或是必须从被调用的程序中重新计算出这些返回的地址。

7.5.4 名字传递

这是传递机制中最复杂的参数了。由于名字传递的思想是直到在被调用的程序真正使用了自变量 (作为一个参数) 之后才对这个自变量赋值，所以它还称作延迟赋值 (delayed evaluation)。因此，自变量的名称或是它在调用点上的结构表示取代了它对应的参数的名字。例如在代码

```
void p(int x)
{ ++x; }
```

中，若做出了一个如 p(a[i]) 的调用时，其结果是计算 ++(a[i])。因此，若在 p 中使用 x 之前改变 i，那么它的结果就与在引用传递或在值结果传递中的不同。例如，在代码 (C 语法)

```
int i;
int a[10];

void p(int x)
{ ++i;
  ++x;
}

main()
{ i = 1;
  a[1]=1;
```

```
a[2]=2;  
p(a[1]);  
return 0;  
}
```

对p的调用的结果是将a[2]设置为3并保持a[1]不变。

名字传递的解释如下：在调用点上的自变量的文本被看作是它自己右边的函数，每当在被调用的过程的代码中到达相应的参数名时，就要计算它。我们总是在调用程序的环境中计算自变量，而总是在过程的定义环境中执行过程。

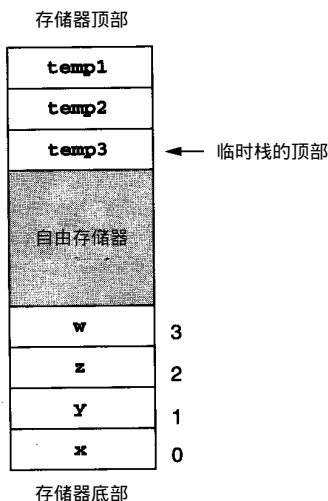
在语言Algo160中，名字传递是作为一种参数传递机制（连同值传递）提供的，但是由于几个原因却变得并不流行。首先在副作用方面，它的结果会令人吃惊（正如前一个例子所示）。其次，由于每个自变量本质上必须被变成过程（有时叫做挂起或形实转换程序），每当计算自变量时都须调用该过程，所以在实现上是有一定困难的。再者，由于它不仅将简单的自变量计算转变为过程调用而且还引起多个赋值，所以它的效率并不高。这种机制的一个变形称作懒惰赋值（lazy evaluation），它最近在纯的函数语言中已较为流行了，在其中通过记忆第一次调用的计算值来（memoizing）阻止重新赋值。因为从未用过的自变量也不会被赋值，所以懒惰赋值可以真正地使实现更为有效。提供懒惰赋值作为传递机制的参数的语言是 Miranda和Haskell。读者可查看“引用与参考”一节获得更多的信息。

7.6 TINY语言的运行时环境

在本章的最后一节，我们描述 TINY语言运行时环境的结构，所用的示例是一个用于编译的较小的简单语言。此时所用的方法与机器无关，读者还可查看下一章中特定机器上实现的示例。

TINY所需的环境比本章所提到过的任何环境都要简单得多。实际上，TINY没有过程，而且它的所有变量都是全局的，因此也就不需要一个活动记录的栈了，而唯一所需的动态存储是在表达式赋值期间临时所用的（在FORTRAN77中，甚至也可将其变成静态的，参见练习）。

TINY环境的一个简单示例是将变量放在程序存储器底部的绝对地址中，并将临时栈分配到顶部。这样，假设有4个变量x、y、z和w，这些变量在存储器底部得到绝对地址0到3，运行时环境在执行中的一个存放了3个临时变量的点上看起来如下所示：



根据体系结构，可能会需要设置一些簿记寄存器以指向存储器的底部和 / 或顶部，接着再使用“绝对的”变量地址作为底部指针的偏移，或是使用存储器的顶部指针作为“临时栈的顶部”指针，或是计算固定顶部指针的临时变量的偏移。当然如果处理器栈是可用的，还有可能就是将其作为临时栈。

为了实现这个运行时环境，TINY编译程序中的符号表必须如最后一章中所描述的那样在存储器中保留变量的地址。这是通过 `st_insert` 函数中提供地址参数以及包括重新得到变量地址的 `st_lookup` 函数完成的(附录B的第1166行到第1171行)：

```
void st_insert( char * name, int lineno, int loc );
int st_lookup ( char * name );
```

此时的语义分析程序必须在第一次遇到变量时就为其赋值，这是通过保留一个初始化到第 1 个地址中的静态存储器地址计数器来实现的(附录B第1413行)：

```
static int location = 0;
```

然后无论何时遇到变量(在读语句、赋值语句或标识符表达式中)，语义分析程序执行代码(附录B第1454行)：

```
if (st_lookup(t->attr.name) == -1)
    st_insert(t->attr.name,t->lineno,location++);
else
    st_insert(t->attr.name,t->lineno,0);
```

当 `st_lookup` 返回 -1 时，变量并不在表中。此时就记录下一个新的地址并添加了位置计数器。另一种情况是变量早已在表中，此时符号表忽略地址参数(并写下0作为一个虚构的地址)。

上面所述内容处理了在 TINY 程序中分配命名了的变量：位于存储器顶部的临时变量的分配以及保留这个分配所需的操作都由代码生成器负责，这将在下一章讨论到。

练习

7.1 为以下的FORTRAN77程序的运行时环境画出一个可能的组织结构，它应与图 7-1相类似。还要保证包括了与对 AVE 的调用时存在的一样的存储器指针。

```
REAL A(SIZE), AVE
INTEGER N, I
10 READ *, N
IF (N.LE.0.OR.N.GT.SIZE) GOTO 99
READ *, (A(I), I=1, N)
PRINT *, 'AVE = ', AVE(A, N)
GOTO 10
99 CONTINUE
END
REAL FUNCTION AVE(B, N)
INTEGER I, N
REAL B(N), SUM
SUM = 0.0
DO 20 I=1, N
20 SUM=SUM+B(I)
AVE = SUM/N
END
```

7.2 为以下的C程序的运行时环境画出一个可能的组织结构，它应与图 7-2相类似。

- a. 在进入函数f中的块A之后。
- b. 在进入函数g中的块B之后。

```

int a[10];
char * s = "hello";

int f(int i, int b[])
{ int j=i;
  A:{ int i=j;
      char c = b[i];
      ...
    }
  return 0;
}

void g(char * s)
{ char c = s[0];
  B:{ int a[5];
      ...
    }
}

main()
{ int x=1;
  x = f(x,a);
  g(s);
  return 0;
}

```

- 7.3 在对factor的第2次调用之后为程序清单 4-1中的C程序的运行时环境画出一个可能的组织结构，假设输入串为(2)。
- 7.4 为以下的Pascal程序画出活动记录的栈，并在对过程c的第2次调用之后表示出控制和访问链。描述如何在c中访问变量x。

```

program env;

procedure a;
var x:integer ;

  procedure b;
    procedure c;
      begin
        x := 2;
        b;
      end;
    begin (* b *)
      c;
    end;

  begin (* a *)
    b;
  end;

begin (* main *)
  a;
end.

```

- 7.5 为以下的Pascal程序画出活动记录的栈
- 在对p的第1次调用中对a的调用之后。
 - 在对p的第2次调用中对a的调用之后。
 - 程序打印出什么？为什么？

```

program closureEx(output);
var x:integer;

procedure one;
begin
    writeln(x);
end;

procedure p(procedure a);
begin
    a;
end;

procedure q;
var x:integer;
    procedure two;
    begin
        writeln(x);
    end;
begin
    x := 2;
    p(one);
    p(two);
end; (* q *)

begin (*main *)
    x := 1;
    q;
end.

```

- 7.6 考虑以下的Pascal程序。假设一个用户输入包括了3个数1、2和0，则当第1次打印数1画出活动记录的栈。包括所有的控制和访问链以及所有的参数和全局变量，并假设将所有的过程都存储在作为闭包的环境中。

```

program procenv(input,output);

procedure dolist (procedure print);
var x:integer;
    procedure newprint;
    begin
        print;
        writeln(x);
    end;
begin (* dolist *)
    readln(x);
    if x = 0 then begin
        print;
        print;
    end;
end;

```

```

end
else dolist(newprint);
end; (* dolist *)

procedure null;
begin
end;

begin (* main *)
  dolist ( null ) ;
end.

```

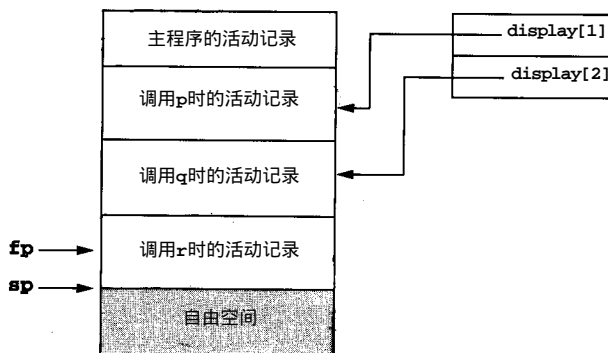
- 7.7 为了完成完整的静态分配，FORTRAN77编译程序需要构造对于程序中的任何表达式计算所需的临时变量的最大数的估计。设计一个方法来估计计算一个表达式所需临时变量的数目，计算通过表达式树的遍历进行。假设表达式是从左到右赋值且必须将每个左子表达式保存在临时变量中。
- 7.8 在允许过程调用中包含可变数量自变量的语言中，找到第 1 个自变量的一种方法是根据 7.3.1 节所述按相反的顺序计算出自变量。
- 按相反顺序计算自变量的另一种方法是识别活动记录，以使用第 1 个自变量即使是在可变数量自变量时也是适用的。描述这样的活动记录组织以及它所需的调用序列。
 - 另一个办法是使用除 *sp* 和 *fp* 之外的另一个称作 *ap* (自变量指针) 的指针。描述使用 *ap* 寻找第 1 个自变量和其所需的调用序列的活动记录结构。
- 7.9 本书讲解了如何处理通过值传递可变长参数 (如开放式数组) (参见例 7.6)，并提出了一个与变长局部变量作用相似的方法。但是当两个变长参数和局部参数都出现时就会有问题了。请利用以下的 Ada 过程作为示例来描述这个问题并提出解决办法：

```

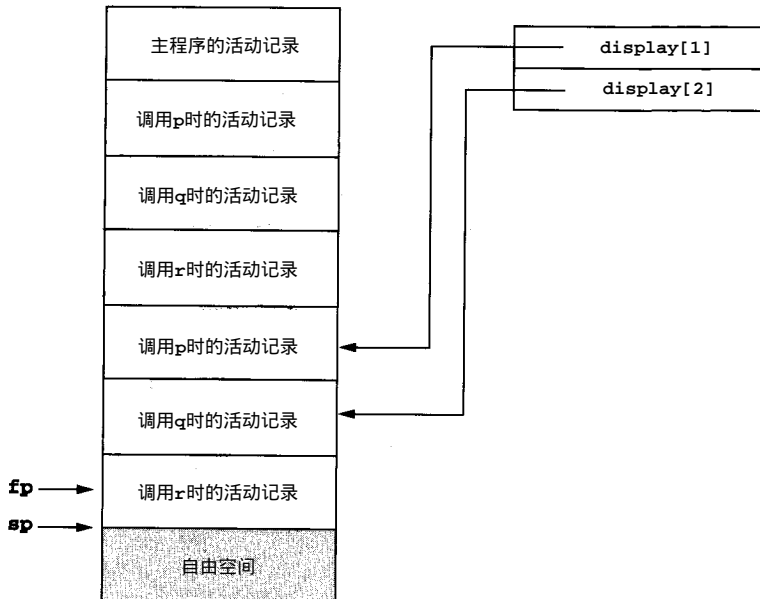
type IntAr is Array(Integer range <>) of Integer;
...
procedure f(x:IntAr; n:Integer) is
  y: Array(1..n) of Integer;
  i: Integer;
begin
  ...
end f;

```

- 7.10 利用局部过程在语言中访问链的另一种方法是由嵌套层在栈外的一个数组中保存访问链。这个数组称作显示 (display)。例如当图 7-7 中的运行时栈带有显示时，应如下所示：



而图7-8中的运行时栈如下所示：



- 描述显示如何可以从很深嵌套的过程中提高非局部引用的效率。
- 使用显示重做练习7.4。
- 描述实现显示必需的调用序列。
- 利用过程参数在一个语言中使用显示出现了一个问题。利用练习7.5描述问题。

7.11 考虑C语法中的以下过程：

```
void f( char c, char s[10], double r )
{ int * x;
  int y[5];
  ...
}
```

- 使用标准C参数传递约定，并假设数据大小为：整型 = 2个字节，字符 = 1个字节，双精度 = 8个字节，地址 = 4个字节，利用本章所描述的活动记录结构判断以下的fp的偏移：(1) c. (2) s[7]. (3) y[2].
- 假设所有的参数都由值(包括数组)传递，重复a。
- 假设所有的参数都引用传递，重复a。

7.12 执行以下的C程序并用运行时环境解释其输出：

```
#include <stdio.h>

void g(void)
{ {int x;
  printf("%d\n",x);
  x = 3;}
  {int y;
  printf("%d\n",y);}
}
```

```
int* f(void )
{ int x;
  printf("%d\n",x);
  return &x;
}

void main()
{ int *p;
  p = f();
  *p =1;
  f();
  g();
}
```

7.13 为以下的C++类画出对象的存储器框架以及如7.4.2节所述的虚拟函数表：

```
class A
{ public:
  int a;
  virtual void f();
  virtual void g();
};
class B : public A
{ public :
  int b;
  virtual void f();
  void h();
};
class C : public B
{ public:
  int c;
  virtual void g();
}
```

7.14 在面向对象的语言中的虚拟函数表为一个方法而保存层次图搜索遍历，但这是有代价的。请解释这个代价是什么。

7.15 利用7.5节中所谈到的4个参数传递办法给出以下程序的输出(用C语法编写)：

```
#include <stdio.h>
int i=0;

void p(int x, int y)
{ x += 1;
  i += 1;
  y += 1;
}

main()
{ int a[2]={1,1};
  p(a[i],a[i]);
  printf("%d %d\n",a[0],a[1]);
  return 0;
}
```

7.16 利用7.5节中所谈到的参数传递办法给出以下程序的输出(在C语法中)：

```
#include <stdio.h>
int i=0;

void swap(int x, int y)
{ x = x + y;
  y = x - y;
  x = x - y;
}

main()
{ int a[3] = {1,2,0};
  swap(i,a[i]);
  printf("%d %d %d %d\n",i,a[0],a[1],a[2]);
  return 0;
}
```

7.17 假设将FORTRAN77的子例程p说明如下：

```
SUBROUTINE P(A)
  INTEGER A
  PRINT *, A
  A = A + 1
  RETURN
END
```

且从主程序中调用如下：

```
CALL P(1)
```

在某些FORTRAN77系统中，这将导致一个运行时的错误。而在其他系统中却不会发生运行时错误，但若用1作为其自变量而再调用一次子例程，它将会打印出值2。请根据运行时环境解释这两个行为是如何发生的。

7.18 名字传递的一个变形称作由文本传递 (pass by text)，其中的自变量的赋值是用的延迟风格，这与名字传递相同，但每个自变量都是在被调用的过程的环境中而不是在调用的环境中赋值。

- a. 说明由文本传递的结果可与名字传递不同。
- b. 描述一个运行时环境组织以及可被用作实现由文本传递的调用序列。

编程练习

- 7.19 正如在7.5节中所描述的一样，名字传递或延迟赋值都可被看作是将自变量包在一个函数体中(或中止)，它在参数每次出现在代码中时被调用。重写练习7.16的C代码以在这个风格中实现swap函数的参数，并证实该结果确实是与由名字传递相等。
- 7.20 a. 正如在7.5.4节中所述，名字传递中的一个有效的实现可以通过记忆它第1次赋值的自变量的值。重写前一个练习中的代码以实现这样的记忆，并比较两个练习的结果。
b. 记忆会导致与名字传递所不同的结果。请解释它是如何发生的。
- 7.21 可将压缩(7.4.4节)垃圾回收分成两个不同的步骤，并当存储器要求由于缺少足够的大型块而失败时也可由malloc完成。

- a. 重写7.4.3节中的`malloc`过程以包括压缩步骤。
- b. 压缩需要先前分配空间改变的位置信息，这意味着程序必须发现这些改变。描述如何使用指向内存块的指针表来解决这个问题，并重写 a部分的代码以包括这个功能。

注意与参考

FORTRAN77(以及更早的FORTRAN版本)的完全静态环境给出了一个类似汇编环境的原始而直接的环境设计方法。基于栈的环境随着诸如 Algol60的包含递归的语言的出现而流行起来(Naur[1963])。Randell和Russell [1964]详细描述了早期的Algol60基于栈的环境。用于一些C编译器的活动记录组织和调用序列的描述在Johnson和Ritchie [1981]中。用显示代替访问链(练习7.10)在Fisher和LeBlanc[1991]中有详细描述，其中还包括在有过程形式参数的语言中使用时将出现的问题。

动态内存管理在许多数据结构的书中有讨论，比如 Aho,Hopcroft和Ullman [1983]。一个实用的近期浏览在Drozdek和Simon[1995]中给出。`malloc`和`free`的代码实现也是类似的，但Kernighan 和Ritchie[1988]中比7.4.3节的代码稍微简单一些。编译中使用的堆结构设计在Fraser和Hanson [1995]中讨论。

一个垃圾回收的浏览可以在Wilson[1992]或Cohen[1981]中找到。生成的垃圾回收和用于函数语言ML运行时环境在Appel [1992]中描述。Gofer函数语言编译器(Jones[1984])包括标记和清除以及一个两阶段垃圾回收。

Budd[1987]描述一个用于小型Smalltalk系统的完全动态环境，包括继承图的应用和带有引用计数的垃圾回收器。C++中使用的虚拟函数表在Ellis和Stroustrup[1990]中描述，还包括处理多继承的扩展。

更多的参数传递技术可以在Louden[1993]中找到，其中还有懒惰赋值的描述。懒惰赋值的实现技术可以在Peyton Jones[1987]中找到。

第8章 代码生成

本章要点

- 中间代码和用于代码生成的数据结构
- 基本的代码生成技术
- 数据结构引用的代码生成
- 控制语句和逻辑表达式的代码生成
- 过程和函数调用的代码生成
- 商用编译器中的代码生成：两个案例研究
- TM：简单的目标机器
- TINY 语言的代码生成器
- 代码优化技术考察
- TINY 代码生成器的简单优化

在这一章中，我们着手编译器的最后工作——用来生成目标机器的可执行代码，这个可执行代码是源代码语义的忠实体现。代码生成是编译器最复杂的阶段，因为它不仅依赖于源语言的特征，而且还依赖于目标结构、运行时环境的结构和运行在目标机器的操作系统的细节信息。通过收集源程序进一步的信息，并通过定制生成代码以便利用目标机器，如寄存器、寻址模式、管道和高速缓存的特殊性质，代码生成通常也涉及到了一些优化或改善的尝试。

由于代码生成较复杂，所以编译器一般将这一阶段分成几个涉及不同中间数据结构步骤，其中包括了某种称做中间代码 (intermediate code) 的抽象代码。编译器也可能没有生成真正的可执行代码，而是生成了某种形式的汇编代码，这必须由汇编器、链接器和装入器进行进一步处理。汇编器、链接器和装入器可由操作系统提供或由编译器自带。在这一章中，我们仅仅集中关注于中间代码和汇编代码的生成，这两者之间有很多共同特性。我们不考虑汇编代码到可执行代码的更进一步的处理，汇编语言或系统的编程文本可以更充分地处理它。

本章的第1节考虑中间代码的两种普遍形式，三地址码和 P-代码，并且讨论它们的一些属性。第2节描述生成中间代码或汇编代码的基本算法。接下来的章节讨论针对不同语言特性的代码生成技术，这包括了表达式、赋值语句、控制语句 (如 if 语句，while 语句) 以及过程和函数调用。

之后的一节将应用在前面章节中学到的技术开发 TINY 语言的一个汇编代码生成器。由于在这种细节水平上的代码生成需要实际的目标机器，因此首先讨论一个目标结构和机器模拟器 TM。附录 C 提供了源代码清单。然后，我们再描述完整的 TINY 语言的代码生成器。最后给出一个关于标准代码的改善、优化技术的简介，同时描述了怎样将一些简单的技术融入到 TINY 代码生成器之中。

8.1 中间代码和用于代码生成的数据结构

在翻译期间，中间表示 (intermediate representation) 或 IR 代表了源程序的数据结构。迄今为止，本文使用了抽象语法树作为主要的 IR。除 IR 外，翻译期间的主要数据结构是符号表，这在第6章中已学过了。

虽然抽象语法树是源代码完美充分的表述，即使对于代码生成也不过这样 (这一点我们将在后面的章节中看到)，但是它与目标代码极不相像，在控制流构造的描述上尤为如此。在控制流构造上，目标代码 (如机器代码或汇编代码) 使用转移语句而不是 if 和 while 语句。因此，

编译器编写者可能希望从语法树生成一个更接近目标代码的中间表示形式，或者用这样一个中间表示代替语法树，然后再从这个新的中间表示生成目标代码。这种类似目标代码的中间表示称为中间代码(intermediate code)。

中间代码能采用很多形式，几乎有多少种编译器就有多少种中间代码形式。然而所有中间代码都代表了语法树的某种线性化(linearization)形式，也就是说，语法树用顺序形式表示。中间代码可以是高水平的，它几乎和语法树一样可以抽象地表示各种操作。它或者还可以非常接近目标代码。它可以使用或不使用目标机器和运行时环境的细节信息，如数据类型的尺寸、变量的地址和寄存器。它可以混合或不混合符号表中包括的信息，如作用域、嵌套层数和变量的偏移量。假如它混合了符号表中包括的信息，目标代码的生成基于中间代码就足够了；否则，编译器必须保留符号表。

当编译器的目标是产生非常高效的代码时，中间代码是极其有用的。如要产生高效的代码就需要相当数量的目标代码属性分析，使用中间代码能使这变得容易。特别地，虽然从语法树中直接得到混合细节分析信息的附加数据结构不是不可能的，但它能更容易地从中间代码中得到。

中间代码在使编译器更容易重定向上也是有用的：假如中间代码与目标机器相对独立，那么要为不同目标机器生成代码就仅需重写从中间代码到目标代码的翻译器。这比重写整个编译器要容易。

在本节中我们将学习两个中间代码的普遍形式：三地址码(three-address code)和P-代码(P-code)。这两种中间代码以许多不同的形式出现，我们的研究将仅集中在普遍特性上，而不是代表了某一个版本的细节化描述。这种描述能在本章最后“注意与参考”节所描述的文献中找到。

8.1.1 三地址码

三地址码最基本的用法说明被设计成表示算术表达式的求值，形式如下：

$$x = y \text{ op } z$$

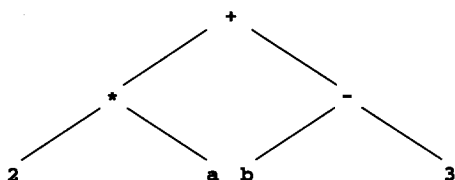
这个用法说明表示了对 y 和 z 的值的应用操作符 op ，并将值赋给 x 。这里的 op 可以是算术运算符，如+或-，也可以是其他能操作于 y 、 z 值的操作符。

三地址码这个名字来自于这个用法说明的形式，因为 x 、 y 、 z 通常代表了内存中的3个地址。但是要注意， x 的地址的使用不同于 y 、 z 的地址的使用。 y 、 z (x 不能)可以代表常量或没有运行时地址的字面常量。

为了看清这种形式的三地址码如何能表示表达式的计算，考虑下边的算术表达式

$$2 * a + (b - 3)$$

语法树如下：



相应的三地址码如下

$$t1 = 2 * a$$

```
t2 = b - 3
t3 = t1 + t2
```

三地址码要求编译器产生临时变量名，在这个例子中的是 `t1`、`t2`和`t3`。这些临时变量对应于语法树的内部节点而表示计算值，在这个例子中用临时变量 `t3`表示根节点值^①。这里并没有说明如何在内存中分配这些临时变量；它们通常将被分到寄存器中，但也有可能保存在活动记录里面(参见第7章“临时栈”的讨论)。

三地址码仅代表了从左至右的语法树线性化，因为首先列出了相对于根的左子树的求值的代码，编译器在某种情况下希望用另一种顺序也是有可能的。我们注意到对于三地址码来说，是有可能使用另一顺序，即(临时变量有不同的意思)：

```
t1 = b-3
t2 = 2*a
t3 = t2+t1
```

很明显，上面所示的这种三地址码形式对于表示所有语言，即使是最小的程序语言的特性也是不够的。例如一元操作符(如负号)就需要一个三地址码的变种(包含两个地址)如：

```
t2 = -t1
```

为适应标准程序语言的使用结构，必须为每个结构改变三地址码的形式。如果语言中含有不常见到的特性，那么就必须为表达的这种特性发明另一种三地址码形式。这就是三地址码之所以没有标准形式的原因(正如语法树没有标准形式一样)。

在这一章余下的章节中我们将逐个处理一些程序语言共有的结构，并显示怎样将这些结构翻译成三地址码。为了知道其结果，我们给出一个用 TINY语言编写的完整示例。

请考虑来自第1章1.7节的TINY例子，该例计算了一个整数的阶乘，我们把它重新放在程序清单8-1中。

程序清单8-1 TINY程序示例

```
{ Sample program
  in TINY language--
  computes factorial
}
read x; { input an integer }
if 0 < x then { don't compute if x <= 0 }
  fact := 1;
  repeat
    fact := fact * x;
    x := x - 1
  until x = 0;
  write fact { output factorial of x }
end
```

程序清单8-2是这个例子的三地址码。这个代码包含了许多三地址码的不同形式。首先，内置的输入和输出操作符 `read`和`write`已被直接翻译成一地址指令。其次，这里有一个条件转移指令 `if_false`，它通常被用来翻译 `if`语句和循环语句，它包含两个地址：被检测的条

^① 诸如 `t1`、`t2`的名字仅仅意味着是这种代码通常类型的代表。实际上，如果像这里一样使用源代码名字的话，三地址码中的临时变量名必须区别于在实际的源代码中所用的名字。

件值和转移的代码地址。一地址的 `label` 指令指示了这个转移地址的位置。有了用以实现三地址码的数据结构, 这些 `label` 指令可能并不是必需的。 `halt` 指令(无地址)用来标志代码的结束。

程序清单 8-2 程序清单 8-1 中 TINY 程序的三地址码

```
read x
t1 = x > 0
if_false t1 goto L1
fact = 1
label L2
t2 = fact * x
fact = t2
t3 = x - 1
x = t3
t4 = x == 0
if_false t4 goto L2
write fact
label L1
halt
```

最后, 我们注意到原代码中的赋值语句导致了如下形式的 `copy` 指令

```
x=y
```

例如, 例程中语句

```
fact:=fact*x;
```

翻译成 2 个三地址码

```
t2=tact *x
fact=t2
```

即使三地址码指令也是足够了, 这种情况的技术原因将在 8.2 节中讲述。

8.1.2 用于实现三地址码的数据结构

三地址码通常不被实现成我们所写的文本形式(虽然这是可能的), 相反是将其实现为包含几个域的记录结构。并将整个三地址指令序列实现成链表或数组, 它能被保存在内存中并在需要时可以从临时文件中读写。

最通常的实现是将三地址码按其所显示的内容实现。这意味着有 4 个域是必需的: 1 个操作符和 3 个地址。对于那些少于 3 个地址的指令, 将一个或更多的地域置成 `null` 或 “empty”, 具体选择哪个域取决于实现。必须有 4 个域的三地址码表示叫做四元式 (quadruple)。程序清单 8-2 的三地址码的四元式在程序清单 8-3 中给出。这里我们用数学中的元组概念书写四元式。

程序清单 8-3 程序清单 8-2 中的三地址码的四元式实现

```
(rd,x,_,_)
(gt,x,0,t1)
(if_f,t1,L1,_)
(asn,1,fact,_)
(lab,L2,_,_)
(mul,fact,x,t2)
```

```
(asn,t2,fact,_)
(sub,x,1,t3)
(asn,t3,x,_)
(eq,x,0,t4)
(if_f,t4,L2,_)
(wri,fact,_,_)
(lab,L1,_,_)
(halt,_,_,_)
```

程序清单8-4 程序清单8-3中四元式的数据结构的C定义

```
typedef enum {rd,gt,if_f,asn,lab,mul,
              sub,eq,wri,halt,...} OpKind;
typedef enum {Empty,IntConst,String} AddrKind;
typedef struct
{ AddrKind kind;
  union
  { int val;
    char * name;
  } contents;
} Address;
typedef struct
{ OpKind op;
  Address addr1,addr2,addr3;
} Quad;
```

程序清单8-4所示的是程序清单8-3中的四元式的C类型定义，在这些定义中，允许地址为整数、常量或字符串(代表临时变量或一般变量的名字)。由于使用了名字，就必须将其加入到符号表中，以供进一步处理时查询。另一种替代方法是在四元式中使用指向符号表入口的指针，这将避免额外的查询。这对可嵌套的语言来说有特别的好处，因为这时候的名字查询还需要更多的嵌套层信息，如果常量也输入符号表，那么将不再需要地址数据类型中的 union。

三地址码另一个不同的实现是用自己的指令来代表临时变量，这样地址域从 3个减少到了两个。因此在三地址指令中包含3个地址而目标地址总是一个临时变量^①。如此的三地址码实现称为三元式(triple)。它要求：或是通过数组的索引号或是通过链表指针，每个三地址指令都是可引用的，如程序清单8-5是程序清单8-2的三地址码作为三元式实现的抽象表达。在那幅图中，我们使用了一个数字系统，它对应于代表三元式的数组索引。在三元式内，把三元式引用用圆括号括起，以同常量相区别。程序清单8-5取消了label指令，代之以三元式引用。

程序清单8-5 程序清单8-2中三地址码的三元式表示

```
(0) (rd,x,_)
(1) (gt,x,0)
(2) (if_f,(1),(11))
(3) (asn,1,fact)
(4) (mul,fact,x)
(5) (asn,(4),fact)
(6) (sub,x,1)
(7) (asn,(6),x)
```

① 这不是三地址码的固有属性，但能通过实现来保证。例如，程序清单8-2的代码就是这样的(程序清单8-3也是)。

```
(8)    (eq,x,0)
(9)    (if_f,(8),(4))
(10)   (wri,fact,_)
(11)   (halt,_,_)
```

三元式是代表三地址码的有效方法，空间数量减少了且编译器不需要产生临时变量名；然而，三元式也有一个不利因素：用数组索引代表三元式使得三元式位置的移动变得很困难，而如用链表的话就不存在这个问题。三元式和对三元式的C代码定义的问题仍处于实践阶段。

8.1.3 P- 代码

在70年代和80年代早期，P-代码作为由许多Pascal编译器产生的标准目标汇编代码被设计成称作P-机器(P-machine)的假想栈机器的实际代码。P-机器在不同的平台上由不同的解释器实现。这个思想使得pascal编译器变得容易移植，只需对新平台重写P-机器解释器即可。P-代码已被证明是一个非常有用的中间代码，它的各种扩展和修改版在许多自然代码的编译器中得到了使用，其中大多数都是针对类Pascal语言的。

由于将P-代码设计成直接可执行的，所以它包含了对特殊环境的明确描述、数据尺寸，还有P-机器大量的特有信息，如果要理解P-代码程序，就必须提供上述信息。为避开细节而恰当地说明问题，在这里只描述P-代码的一个简化的抽象版本。各种不同版本的实际P-代码的描述能在本章最后列出的大量参考书中找到。

从我们的目的出发，P-机器包括一个代码存储器、一个未指定的存放命名变量的数据存储器、一个存放临时数据的栈，还有一些保持栈和支持执行的寄存器。作为P-代码的第1个例子，考虑如下表达式，这个表达式在8.1.1节中已用过，它的语法树在8.1.1节：

$$2*a+(b-3)$$

这个表达式的P-代码版本如下：

```
ldc 2          ; load constant 2
lod a          ; load value of variable a
mpi           ; integer multiplication
lod b          ; load value of variable b
ldc 3          ; load constant 3
sbi           ; integer subtraction
adi           ; integer addition
```

这些指令被看作代表如下的P-机器操作：ldc 2首先将值2压入临时栈，然后，lod a将变量a的值压入栈。指令mpi将这两个值从栈中弹出，使之相乘（按弹出的相反顺序），再将结果压入栈。接下来两个指令(lod a 和 ldc 3)将b的值和常量3压入栈（现在栈中有3个值），随后，sbi指令弹出栈顶的两个值，用第1个值去减第2个值，再把结果压入栈中，最后adi指令弹出余下的两个值并使之相加，再将结果压入栈。代码结束时，栈中只有一个值，它代表了这次运算的结果。

作为第2个例子，考虑赋值语句：

$$x := y + 1$$

对应于如下的P-代码指令：

```
lda x          ; load address of x
lod y          ; load value of y
```



```
ldc 1          ; load constant 1
adi            ; add
sto            ; store top to address
              ; below top & pop both
```

注意，这段代码首先计算x的地址，然后将表达式的值赋给x，最后执行sto命令，这个命令需要临时栈顶上的两个值：一个是要被存储的值，在它下面的那个是值所要存入的地址。sto指令也弹出两个值(在这例子中使栈变空)。在x:=y+1中，左边的x的用处和右边的y的用处不一样，相应的P-代码用装入地址(lda)和装入值(lod)作出区别。

作为本节中最后的一个P-代码例子，我们对程序清单8-1中的TINY程序给出P-代码的翻译，如程序清单8-6所示，其中每条操作都有注释。

程序清单8-6 程序清单8-1中TINY程序的P-码指令

```
lda x          ; load address of x
rdi            ; read an integer, store to
              ; address on top of stack (& pop it)
lod x          ; load the value of x
ldc 0          ; load constant 0
grt            ; pop and compare top two values
              ; push Boolean result
fjp L1         ; pop Boolean value, jump to L1 if false
lda fact       ; load address of fact
ldc 1          ; load constant 1
sto            ; pop two values, storing first to
              ; address represented by second
lab L2         ; definition of label L2
lda fact       ; load address of fact
lod fact       ; load value of fact
lod x          ; load value of x
mpi            ; multiply
sto            ; store top to address of second & pop
lda x          ; load address of x
lod x          ; load value of x
ldc 1          ; load constant 1
sbi            ; subtract
sto            ; store (as before)
lod x          ; load value of x
ldc 0          ; load constant 0
equ            ; test for equality
fjp L2         ; jump to L2 if false
lod fact       ; load value of fact
wri            ; write top of stack & pop
lab L1         ; definition of label L1
stp
```

程序清单8-6中的P-代码包含了几个新的P-代码指令。首先，无参的rdi和wri指令实现了TINY中的整型的read和write语句。rdi P-代码指令要求要读的那个变量地址应位于栈顶，这个地址作为指令的一部分被弹出。wri指令要求要写的值地址位于栈顶，这个值作为指令的一部分弹出。程序清单8-6中出现的另一些新指令是lab指令，它定义了标签名字的位置；fjp指令(“false jump”)需要在栈顶有一个布尔值；sbi指令(整数减法)的操作类似于其他算术指令；grt指令(大于)和etu(等于)指令需要两个整型值位于栈顶(要被弹出)然后压入他们的布尔

型结果。`stp`指令对应于前面三地址码的`halt`指令。

1) **P-代码和三地址码的比较** P-代码在许多方面比三地址码更接近于实际的机器码。P-代码指令也需要较少地址；我们已见过的都是一地址或零地址指令，另一方面，P-代码在指令数量方面不如三地址码紧凑，P-代码不是自包含的，指令操作隐含地依赖于栈（隐含的栈定位实际上就是“缺省的”地址），栈的好处是在代码的每一处都包含了所需的所有临时值，编译器不用如三地址码中那样为它们再分配名字。

2) **P-代码的实现** 历史上，P-代码已经大量地作为文本文件生成，但前面的三元地址码的内部数据结构描述（三元式和四元式）也能作用于P-代码的修改版。

8.2 基本的代码生成技术

本节讨论代码生成的基本方法，在下一节，我们将针对单个的语言结构进行代码生成。

8.2.1 作为合成属性的中间代码或目标代码

中间代码生成(或没有中间代码的直接目标代码生成)能被看作是一个属性计算，这类似于第6章中研究的许多属性问题，实际上假如生成代码被看作一个字符串属性（每条指令用换行符分隔），这个代码就成了一个合成属性并能用属性文法定义，并且能在分析期间直接生成或者通过语法树的后序遍历生成。

为了看清楚三地址码或P-代码怎样被作为合成属性定义，考虑下边的文法，它代表了C表达式的子集。

```
exp  id = exp | aexp
aexp aexp + factor | factor
factor ( exp ) | num | id
```

这个文法仅包含了两个操作，赋值(=)和加法(+)[⊖]。记号`id`代表简单标识符，记号`num`代表了表示整数的简单数字序列。这两个记号被假设成有一个预先计算过的`strval`属性，它可以是字符串或词(例如“42”是`num`，“`xtemp`”是`id`)。

1) **P-代码** 我们首先考虑P-代码的情况，由于不需要产生临时变量名，属性文法会简单些，然而，嵌套赋值的存在是一个复杂因素。在这种情况下，我们希望保留被存的值作为赋值表达式的结果值，然而标准的P-代码指令`sto`是有害的，因为所赋的值会丢掉(P-代码在这里显示出了它pascal源，在pascal源代码中不存在嵌套的赋值语句)。我们通过引入一个无害的存储(nondestructive store)指令`stn`来解决这个问题，`stn`和`sto`一样，都假设栈顶有一个值且下面有一地址。`stn`将值存入那个地址，但在丢弃那个地址时栈顶上仍保留了那个值。表8-1是使用这个新的指令后P-代码属性的属性文法。在那幅图中，已经用属性名`pcode`表示P-代码串，并已把两个不同的符号用于串的连接：`++`表示所连的串之间不能在同一行，`||`表示连接一个串用空格相隔。

我们将跟踪某个例子的`pcode`属性计算留给读者并写出来，例如：表达式 $(x=x+3)+4$ 有如下的`pcode`属性：

```
lda x
lod x
```

⊖ 这个例子中的赋值有如下的语义： $x=e$ 将 e 的值存入 x ，该赋值的结果值是 e 。

```
ldc 3
adi
stn
ldc 4
adi
```

表8-1 P-代码合成字符串属性的属性文法

文法规则	语义规则
$exp_1 \quad id = exp_2$	$exp_1.pcode = "lda" \parallel id.strval ++ exp_2.pcode ++ "stn"$
$exp \quad aexp$	$exp.pcode = aexp.pcode$
$aexp_1 \quad aexp_2 + factor$	$aexp_1.pcode = aexp_2.pcode ++ factor.pcode ++ "adi"$
$aexp \quad factor$	$aexp.pcode = factor.pcode$
$factor \quad (exp)$	$factor.pcode = exp.pcode$
$factor \quad num$	$factor.pcode = "ldc" \parallel num.strval$
$factor \quad id$	$factor.pcode = "lod" \parallel id.strval$

2) 三地址码 前面那个简单表达式的三地址码属性文法在表 8-2 中给出。在那张表中，我们称代码属性为 *tacode*。同表 8-1，也用 ++ 表示其间插有换行符的串连接，|| 表示其间有空格的串连接。与 P-代码不同，三地址码要求为表达式的中间结果生成临时变量名，这就要求属性文法在每个节点中都包括一个新名字属性。这个属性也是合成的，如果没有为一个内部节点分配一个新产生的临时名，就用 *newtemp()* 产生一个临时名字系列 *t1*、*t2*、*t3*，...（每次调用 *newtemp()* 就返回一个新的）。在这个简单例子中，仅对应于 + 的节点需临时名，赋值操作使用右边的表达式的名字。

表8-2 三地址码作为合成串属性的属性文法

文法规则	语义规则
$exp_1 \quad id = exp_2$	$exp_1.name = exp_2.name$ $exp_1.tacode = exp_2.tacode ++ id.strval \parallel "=" \parallel exp_2.name$
$exp \quad aexp$	$exp.name = aexp.name$ $exp.tacode = aexp.tacode$
$aexp_1 \quad aexp_2 + factor$	$aexp_1.name = newtemp()$ $aexp_1.tacode = aexp_2.tacode ++ factor.tacode$ $++ aexp_1.name \parallel "=" \parallel aexp_2.name$ $\parallel "+" \parallel factor.name$
$aexp \quad factor$	$aexp.name = factor.name$ $aexp.tacode = factor.tacode$
$factor \quad (exp)$	$factor.name = exp.name$ $factor.tacode = exp.tacode$
$factor \quad num$	$factor.name = num.strval$ $factor.tacode = ""$
$factor \quad id$	$factor.name = id.strval$ $factor.tacode = ""$

表8-2的产生式 $exp \quad aexp$ 和 $aexp \quad factor$ 中，将名字属性即 *tacode* 属性从子节点提到父节点，在操作符内部节点中，新名字属性应在联合的 *tacode* 代码之前生成。在叶产生式 $factor \quad num$ 和

factor *id* 中记号的串值记作 *factor.name*。与 P-代码不同, 在叶产生式的节点上没有生成三地址码(用 "" 表示空串)。

再次, 我们让读者按表 8-2 所给出的等式写出表达式 $(x=x+3)+4$ 的每一步的 *tacode*, 这个表达式的 *tacode* 属性如下:

```
t1 = x+3
x = t1
t2 = t1+4
```

(这里假设 *newtemp*() 用后序调用并且产生从 *t1* 开始的临时名)。注意 $x=x+3$ 是怎样用临时名产生两个三地址指令的。这是属性值总是为每一个子表达式产生一个临时名的结果, 它包括了赋值号的右边部分。

将代码生成看成一个合成字符串属性计算, 对于清楚地显示语法树各部分代码系列的关系以及比较不同的代码生成方法是很有用的, 但它作为真实的代码生成技术是不实际的, 这有几个原因: 首先, 串连接的使用造成了过度的串拷贝因而浪费了内存 (除非连接符做得非常复杂), 其次, 通常希望产生几片代码作为代码产生的收益并且将这几片代码写入一个文件或将它们插入一个数据结构 (如四元式数组), 这就需要语义动作, 而语义动作又不与属性的标准后序合成有牵连。最后, 即使将代码看作是纯合成是有用的, 但通常的代码生成很大程度上依赖于继承属性, 这将使得属性文法大大复杂化。由于这个原因, 我们就不必麻烦再去写出实现前面例子中的属性文法的代码了 (即使是伪码)。相反地, 在下一节中, 我们将转向更直接的代码生成技术。

8.2.2 实际的代码生成

标准的代码生成或者涉及语法树后序遍历的修改, 这棵语法树是由前面例子的属性文法所包含的, 或者如没有显式生成的语法树, 则在分析中涉及了相等效的动作。基本算法由下面的递归过程描述 (用于二叉树, 但容易将其推广到节点子树多于 2 的情况):

```
procedure genCode ( T:treenode );
begin
  if T is not nil then
    generate code to prepare for code of left child of T;
    genCode (left child of T);
    generate code to prepare for code of right child of T;
    genCode (right child of T);
    generate code to implement the action of T;
end;
```

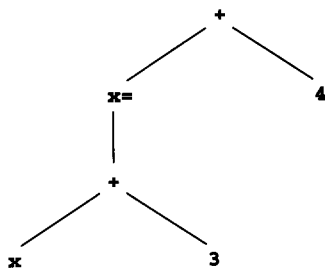
注意, 这个递归遍历过程不仅有一个后序部分 (产生实现 *T* 动作的代码), 而且还有一个前序和一个中序部分 (为 *T* 的左右子树产生准备代码)。通常, *T* 表示的每一个动作需要前序和中序准备代码的稍微不同的一个版本。

为了详细地看清在一个特殊的例子中怎样构造产生代码的过程, 考虑我们在这一节已用过的简单算术表达式的语法, 这个语法的抽象语法树的 C 语言定义如下所示:

```
typedef enum { Plus, Assign } Optype;
typedef enum { OpKind, ConstKind, IdKind } NodeKind;
```

```
typedef struct streenode
{
    NodeKind kind;
    Otype op; /* used with OpKind */
    struct streenode *lchild, *rchild;
    int val; /* used with ConstKind */
    char * strval;
    /* used for identifiers and numbers */
} STreeNode;
typedef STreeNode *SyntaxTree;
```

用这些定义，表达式 $(x=x+3)+4$ 的语法树如下所示：



注意，赋值节点包含了要被赋于的表达式（在 `strval` 域中），以致一个赋值节点只有一个子树（要被赋于的表达式）[⊖]。

基于这棵语法树的结构，我们能写出一个 `genCode` 过程来产生程序清单 8-7 的 P-代码。对图中的代码作如下注释：首先，代码用标准 C 函数 `sprintf` 把字符串连接到本地的临时变量 `codestr`。其次，调用过程 `emitCode` 用以生成一个 P-代码行，在数据结构或输出文件中不显示它的细节。最后，两个操作符（加号和赋值号）需要两个不同的遍历顺序，加号仅需要一些后序处理，而赋值需要一些前序处理和一些后序处理。因此不可能在所有情况下，能将递归调用都写成一个样子。

程序清单 8-7 表 8-1 属性文法定义的 P-码的代码生成过程的实现

```
void genCode( SyntaxTree t)
{
    char codestr[CODESIZE];
    /* CODESIZE = max length of 1 line of P-code */
    if (t != NULL)
    {
        switch (t->kind)
        {
            case OpKind:
                switch (t->op)
                {
                    case Plus:
                        genCode(t->lchild);
                        genCode(t->rchild);
                        emitCode("adi");
                        break;
                    case Assign:
                        sprintf(codestr, "%s %s",
                                "lda", t->strval);
                        emitCode(codestr);
                        genCode(t->lchild);
                }
            }
        }
    }
}
```

⊖ 在这个例子中，我们将数字也当作字符串保存在 `strval` 域中。

```

        emitCode("stn");
        break;
    default:
        emitCode("Error");
        break;
    }
    break;
case ConstKind:
    sprintf(codestr,"%s %s","ldc",t->strval);
    emitCode(codestr);
    break;
case IdKind:
    sprintf(codestr,"%s %s","lod",t->strval);
    emitCode(codestr);
    break;
default:
    emitCode("Error");
    break;
}
}
}

```

即使用到了必需的不同顺序的遍历，显示代码生成仍然可以在分析时进行（没有语法树的生成），我们在程序清单 8-8 中出示了一个 Yacc 说明文件，它直接对应于程序清单 8-7 的代码（注意赋值的前序和后序的组合处理是怎样翻译成独立的部分）。

程序清单 8-8 根据表 8-1 的属性文法生成 P-代码的 Yacc 说明

```

%{
#define YYSTYPE char *
    /* make Yacc use strings as values */

    /* other inclusion code ... */
}%

%token NUM ID

%%

exp      : ID
        { sprintf(codestr,"%s %s","lda",$1);
          emitCode(codestr); }
        '=' exp
        { emitCode("stn"); }
        | aexp
        ;

aexp     : aexp '+' factor {emitCode("adi");}
        | factor
        ;

factor   : '(' exp ')'
        | NUM      { sprintf(codestr,"%s %s","ldc",$1);
                     emitCode(codestr); }
        | ID       { sprintf(codestr,"%s %s","lod",$1);

```

```

emitCode(codestr); }

;

%%
/* utility functions ... */

```

请读者按表8-2的属性文法写出一个`genCode`过程和生成三地址码的Yacc说明。

8.2.3 从中间代码生成目标代码

如果编译器或者直接从分析中或者从一棵语法树中产生了中间代码，那么下一步就是产生最后的目标代码(通常在对中间代码的进一步处理之后)，这一步本来就相当复杂，特别是当中间代码为高度象征性的，且只包含了很少或根本没包含目标机器和运行时环境的信息。在这种情况下，最后的代码生成必须支持变量和临时变量的实际定位，并增加支持运行时环境所必需的代码。寄存器的合适定位和寄存器使用信息的维护(如哪个寄存器可用和哪个包含了已知值)是一个特别重要的问题，我们将在本章最后再讨论分配问题细节。现在仅讨论这个处理的通用技术。

通常，来自中间代码的代码生成涉及了两个标准技术：宏扩展(macro expansion)和静态模拟(static simulation)。宏扩展涉及到用一系列等效的目标代码指令代替每一种中间代码指令。这要求编译器保留关于定位和独立的数据结构的代码惯用语的决定，它要求宏过程按照中间代码中涉及的特殊数据的需要改变代码序列。因此，这一步要比在C预处理器或宏汇编器中可利用的宏扩展的简单形式复杂得多。静态模拟包括中间代码效果的直线模拟和生成匹配这些效果的目标代码。这也需要额外的数据结构，它可以是非常简单的跟踪形式，并在与宏扩展的连接中使用，也可以是非常复杂的抽象解释(abstract interpretation)(当计算值时，对它们保持代数学的追踪)。

在考虑从P-代码翻译成三元地址码(反之亦然)时，我们能了解这些技术的细节。想象一个在这节中已作为运行例子用过的小表达式语法，并考虑表达式 $(x=x+3)+4$ ，它的P-代码和三地址码的翻译在前面已经给出。我们首先考虑将这个表达式的P-代码：

```

lda x
lod x
ldc 3
adi
stn
ldc 4
adi

```

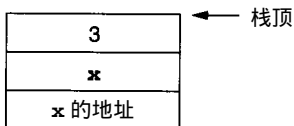
翻译成相应的三地址码：

```

t1 = x + 3
x = t1
t2 = t1 + 4

```

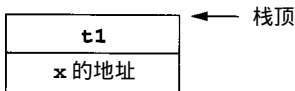
这需要执行一个P-机器栈的静态模拟用以发现代码的三地址码等效式。在翻译期间，用实际的栈数据结构实现它们。在前三条P-代码指令后仍无三地址码指令产生，但是已经将P-机器栈修改以反映这些装入。栈内容如下所示：



现在当处理`adi`操作时，产生三地址指令

```
t1=x+3
```

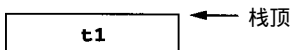
栈改成：



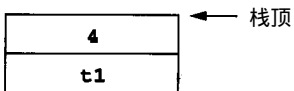
`stn`指令造成三地址指令

```
x=t1
```

生成，栈变成为：



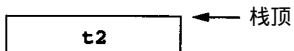
下一个指令压常量4入栈：



最后`adi`指令生成三地址指令

```
t2 = t1 + 4
```

栈变成为



这就完成了静态模拟和翻译。

我们现在考虑将三地址码翻译成P-代码的情况。假如忽略临时变量名增加的复杂性，就能用简单的宏扩展来完成。因此，一个三地址指令

```
a = b + c
```

总能被翻译成如下的P-代码指令序列

```
lda a
lod b ; or ldc b if b is a const
lod c ; or ldc c if c is a const
adi
sto
```

这导致了如下的三地址码到P-代码的翻译(有点不令人满意)

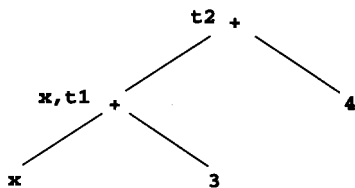
```
lda t1
lod x
ldc 3
adi
sto
lda x
lod t1
sto
lda t2
```

```

lod t1
ldc 4
adi
sto

```

假如要消除额外的临时变量，就需要用一个比宏扩展复杂得多的方案。一种可能是从三地址码生成一棵新树，这棵树用每个指令的操作符和所分配的名字来标记树节点，从而显示代码的效果。它可看作是静态模拟的一种形式。前面的三地址码的结果树如下：



注意三地址指令

```
x = t1
```

在树中不生成节点，而是造成 `t1` 节点获得另一个名字 `x`。这棵树同源代码的语法树类似，但又有不同^①。P-代码能从这个树产生，这非常类似于从语法树中产生 P-代码。通过对内部节点只分配永久名，而消除了临时变量。因此，在这棵样例树中，只分配了 `x`，在 P-代码中根本没有使用 `t1`、`t2`。对应于根节点 (`t2`) 的值被放在 P 机器栈中。这导致了同前面完全一样的 P-代码生成，只要在存储时用 `stn` 代替 `sto` 即可。我们鼓励读者编写伪码或 C 代码执行这个处理。

8.3 数据结构引用的代码生成

8.3.1 地址计算

在前一节中，我们已经看到如何生成一个简单的算术表达式和赋值语句的中间代码。这些例子中，所有的基本值或者是常量或者是简单变量（程序变量如 `x`，临时变量如 `t1`）。简单变量仅通过名字识别——到目标代码的翻译需要这些名字由实际地址代替，这些地址可以是寄存器、绝对地址（针对全局变量），或是活动记录的偏移量（针对局部变量，可能包括嵌套层）。这些地址可以在中间代码生成时插入，也可以推迟到实际代码生成时（用符号保持地址）。

为了定位实际地址，有许多情况需要执行地址计算，即使是在中间代码中，这些计算也必须被直接表达出来。这样的计算发生在数组下标记录域和指针引用中。我们将依次讨论这些情况。但首先将描述一下可以表达这样的地址计算的三地址码和 P-代码扩展。

1) 用于地址计算的三地址码 在三地址码中，对新操作符的需要不是很多。通常的算术操作符能被用来计算地址，但对于显示地址模式的方法需要几个新符号。在我们的三地址码版本中，使用了与 C 语言中意义一样的“&”和“*”来指示地址模式。例如，假设想把常量 2 存放在变量 `x` 加上 10 个字节的地址处，用三地址码表示如下：

```

t1 = &x + 10
*t1 = 2

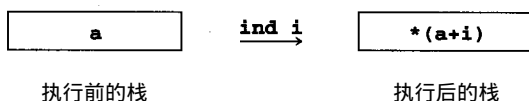
```

^① 这棵树是称为基本块的 DAG(DAG of a basic block)的一般结构的特例，这在 8.9.3 节中描述。

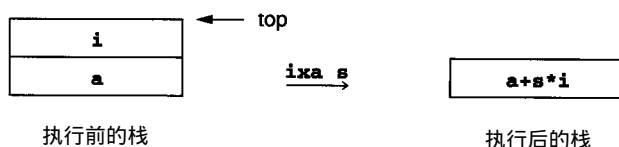
这些新地址模式的实现要求三地址码的数据结构包含一个或多个新域。例如，图 8-4 的四元式数据结构增加了一个枚举型的 `AddrMode` 域，枚举值为 `None`、`Address` 和 `Indirect`。

2) 用于地址计算的 **P-代码** 在 P-代码中，通常引入新的指令表示新的地址模式 (由于很少有外在的地址需要指定地址模式)。为了这个目的将引入如下两条指令：

- **ind** (间接装入) 用一个整型偏移量作为参数，假设栈顶有一个地址，就将这个地址与偏移量相加得到新地址，再将新地址中的值压入栈以代替原来栈顶的地址。



- **ixa** (索引地址) 用整型比例因子作为参数，假设一个偏移量已在栈顶并且在其下边有一个基地址，则用比例因子与偏移量相乘，再加上基地址以得到新地址，再将偏移量和基地址从栈中弹出，压入新地址。



这两个 P-代码指令，和以前介绍过的 `lda` (装入地址) 指令，将允许我们执行和三地址码中地址模式一样的地址计算和引用^①。例如，前面的例子 (把常量 2 存入变量 `x` 加 10 字节处的地址) 现在用 P-代码实现如下：

```
lda x
ldc 10
ixa 1
ldc 2
sto
```

我们现在开始讨论数组，记录和指针，随后是目标代码生成和一个扩展的例子。

8.3.2 数组引用

数组引用通过表达式涉及了数组变量下标，得到数组元素的引用或值。正如下面的 C 代码所示：

```
int a[SIZE]; int i, j;
...
a[i+1] = a[j*2] + 3;
```

在这个赋值语句中，`a` 的下标表达式 `i+1` 产生了一个地址 (赋值的目标地址)，而通过下标表达式 `j*2` 在已计算的地址中得到 `a` 元素类型的一个值。由于数值按顺序存放于存储器中，每个地址必须根据 `a` 的基地址 (base address) (即数组 `a` 在存储器中的起始地址) 和线性地依赖于下标的偏移量计算。当需要的是一个值而不是地址时，就必须生成一个额外的间接步骤从已计算的地址中取出值。

① 实际上，`ixa` 指令能用算术运算符来模拟，除了在 P-代码中，这些操作被类型化 (`adi` = 整数相乘)，因此不能应用于地址。我们不强调 P-代码的类型限制，因为这涉及额外的参数，为了简化，没有使用。

从下标值中计算偏移量如下：首先，假如下标范围不从 0 开始(这在Pascal和Ada中是可能的)，就必须对下标值作一调整。其次，调整后的下标值必须与比例因子 (scale factor)相乘，比例因子等于存储器中数组元素类型的尺寸。最后比例作用过的下标值加上基地址，从而得到数组元素的最终地址。

例如，C数组引用 $a[i+1]$ 的地址是：[⊖]

```
a + (i + 1) * sizeof (int)
```

更一般地，任何语言中数组元素 $a[t]$ 的地址是：

$$base_address(a) + (t - lower_bound(a)) * element_size(a)$$

我们现在转向用三地址码和P-代码表示这个地址计算的方法。为了不依赖目标机器，假设数组变量的地址就是它的基地址。因此，如果 a 是数组变量，在三地址码和P-代码中， $\&a$ 就是数组 a 的基地址。P-代码

```
lda a
```

将 a 的基地址压入 P-机器栈。由于数组引用计算依赖于目标机器元素类型的尺寸，所以用 $elem_size(a)$ 代表目标机器上数组 a 的元素尺寸[⊙]。由于这是一个静态量(假设是静态类型)，这个表达式在编译时将用常量代替。

1) 数组引用的三地址码 在三地址码中表示数组引用的一个可能的方法是引入两个新的操作。一个是获取数组元素的值

```
t2 = a[t1]
```

还有一个是给数组元素地址中赋值。

```
a[t2] = t1
```

(这些就用符号 $=[]$ 和 $[]=$ 表示)，用这个术语表示实际地址的计算不是必要的(机器依赖性如元素尺寸将从这个概念中消失)。例如，源代码语句

```
a[i+1] = a[j*2] + 3
```

将翻译成下面的三地址指令：

```
t1 = j * 2
t2 = a[t1]
t3 = t2 + 3
t4 = i + 1
a[t5] = t3
```

然而，引入前面所述的地址模式仍是必需的，在处理记录域和指针引用时，用统一方式处理所有地址运算是有意义的，因此在三地址码中也直接写出数组元素地址的计算。例如，赋值语句：

```
t2 = a[t1]
```

能写成：(用更多的临时变量： $t3$ 和 $t4$)

```
t3 = t1 * elem_size(a)
t4 = &a + t3
t2 = *t4
```

⊖ 在C中，一个数组(如此例中的 a)的名字代表了它的基地址。

⊙ 这实际上是一个由符号表提供的函数。

赋值语句

```
a[t2] = t1
```

写成

```
t3 = t2 * elem_size(a)
t4 = &a + t3
*t4 = t1
```

最后，一个更复杂的例子，源代码语句

```
a[i+1] = a[j*2] + 3;
```

翻译成三地址指令如下：

```
t1 = j * 2
t2 = t1 * elem_size(a)
t3 = &a + t2
t4 = *t3
t5 = t4 + 3
t6 = i + 1
t7 = t6 * elem_size(a)
t8 = &a + t7
*t8 = t5
```

2) 数组引用的P-代码 正如前面所描述的，我们用新的地址指令 `ind` 和 `ixa`。指令 `ixa` 实际上由数组地址计算精确地构成。`ind` 指令用来装入前面已计算地址的值（例如实现一个间接装入），数组引用

```
t2 = a[t1]
```

的P-代码表示如下：

```
lda t2
lda a
lod t1
ixa elem_size(a)
ind 0
sto
```

数组赋值

```
a[t2] = t1
```

的P-代码如下：

```
lda a
lod t2
ixa elem_size(a)
lod t1
sto
```

最后，前面那个较复杂的例子

```
a[i+1] = a[j*2] + 3;
```

翻译成P-代码如下：

```
lda a
lod i
```

```

ldc 1
adi
ixa elem_size(a)
lda a
lod j
ldc 2
mpi
ixa elem_size(a)
ind 0
ldc 3
adi
sto

```

3) 数组引用的代码生成过程 这里将显示数组引用是怎样由一个代码生成过程产生的，我们用前一节中的C表达式子集的例子，但扩充了下标操作。要用的新文法如下：

```

exp    subs = exp | aexp
aexp   aexp + factor | factor
factor ( exp ) | num | subs
subs   id | id [ exp ]

```

注意，赋值的目标可能是一个简单变量，也可能是一个有下标的变量（都包括在非终结符 *subs* 中）。我们使用和前面的语法树一样的数据结构，另外为下标增加 *subs* 操作

```

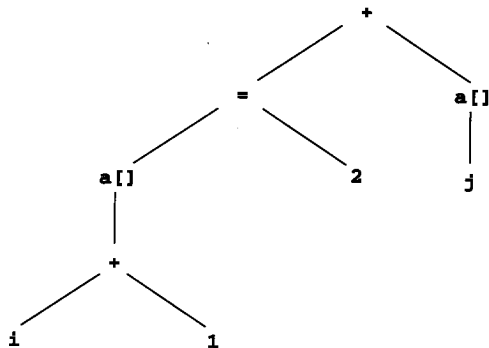
typedef enum { Plus, Assign, Subs } Optype;
/* 像前面一样的其他说明 */

```

由于下标表达式可以在一个赋值号的左边，那么是不可能将目标变量名存入赋值节点中（因为可能没有这样的名字）。赋值节点现在有2个子树，这同加法节点一样。左子树必须是标识符或下标表达式。下标本身只能被用作标识符，因此，存储数组变量名于下标节点。所以，表达式：

```
(a[i+1]=2)+a[j]
```

的语法树如下：



程序清单 8-9是一个为这样的语法树产生 P-代码的代码生成过程(同程序清单 8-7 比照)。这个代码同程序清单 8-7 代码的主要不同在于它需要继承属性 *isAddr*，用于识别下标表达式标识符在赋值号的左边还是在右边。如果 *isAddr* 被设置成 *TRUE*，那么就必须返回表达式的地址；否则返回值。请读者检验下面的表达式 $(a[i+1]=2)+a[j]$ 的 P-代码生成过程：

```

lda a
lod i
ldc 1
adi
ixa elem_size(a)
ldc 2
stn
lda a
lod j
ixa elem_size(a)
ind 0
adi

```

在练习中也请读者检验这个文法的三地址码的代码生成器的构造。

程序清单 8-9 上面的表达式文法对应的 P-码的生成过程的实现

```

void genCode( SyntaxTree t, int isAddr)
{ char codestr[CODESIZE];
  /* CODESIZE = max length of 1 line of P-code */
  if (t != NULL)
  { switch (t->kind)
    { case OpKind:
      { switch (t->op)
        { case Plus:
          if (isAddr) emitCode("Error");
          else { genCode(t->lchild, FALSE);
                  genCode(t->rchild, FALSE);
                  emitCode("adi"); }
          break;
        case Assign:
          genCode(t->lchild, TRUE);
          genCode(t->rchild, FALSE);
          emitCode("stn");
          break;
        case Subs:
          sprintf(codestr, "%s %s", "lda", t->strval);
          emitCode(codestr);
          genCode(t->lchild, FALSE);
          sprintf(codestr, "%s%s%s",
                  "ixa elem_size(", t->strval, ")");
          emitCode(codestr);
          if (!isAddr) emitCode("ind 0");
          break;
        default:
          emitCode("Error");
          break;
      }
      break;
    case ConstKind:
      if (isAddr) emitCode("Error");
      else
      { sprintf(codestr, "%s %s", "ldc", t->strval);
        emitCode(codestr);
      }
    }
  }
}

```



```

        break;
    case IdKind:
        if (isAddr)
            sprintf(codestr, "%s %s", "lda", t->strval);
        else
            sprintf(codestr, "%s %s", "lod", t->strval);
        emitCode(codestr);
        break;
    default:
        emitCode("Error");
        break;
}
}
}

```

4) 多维数组 在数组地址计算中,在大多数语言中多维数组的存在是其复杂的一个因素。例如,在C语言中,二维数组(具有不同的索引大小)可以声明为:

```
int a[15][10];
```

这样的数组可以用部分下标以生成维数更少的数组,或者完全使用下标以产生该数组元素类型的一个值。例如,在C语言中给定上面a的声明,表达式a[i]使用a的部分下标以产生一个一维整型数组,而表达式a[i][j]使用a的全部下标产生一个整型值。数组变量的部分或全部用下标表示后的地址可通过递归使用一维数组中描述的技术来计算。

8.3.3 栈记录结构和指针引用

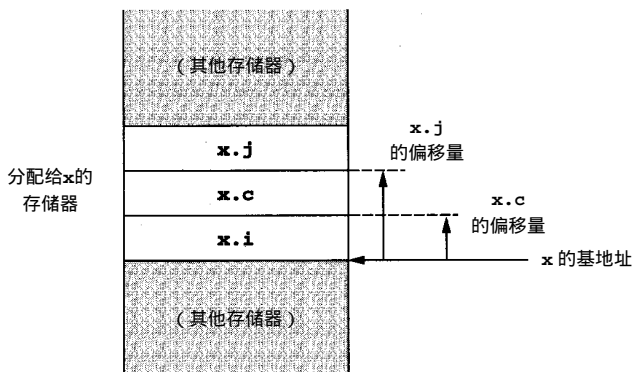
计算记录结构的域地址提出了一个同计算下标数组地址相同的问题。首先,计算结构变量的基地址,然后,找到域偏移量(通常是固定的),两者相加得到结果地址。例如,考虑C语言声明:

```

typedef struct rec
{ int i;
  char c;
  int j;
} Rec;
...
Rec x;

```

一般地,变量x如下图所示那样在存储器中分配,每一个域(i、c和j)有一个从x基地址(可以是它自己在活动记录里的偏移量)开始的偏移量。



注意,域被线性分配(通常从低地址到高地址),每个域的偏移量是常量,第1个($x.i$)偏移量为0,注意:域偏移量依赖于目标机器上不同数据类型的大小,这里没有数组中的比例因子。

为了对记录结构域地址计算编写独立于目标机器的中间代码,必须引入一个新函数返回域的偏移量,并给定结构变量和域名,这个函数称为 `field_offset`,并为它写出两个参数。第1个是变量名,第2个是域名,因此 `field_offset(x,j)` 返回 $x.j$ 的偏移量。和其他相似的函数一样,这个函数能由符号表提供。在任何情况下,这只是编译时的量,因此实际产生的中间代码将用常量代替对 `field_offset` 的调用。

记录结构一般使用指针和动态内存分配来实现动态数据结构(如链表和树),因此将描述指针和域地址计算是如何交互作用的。出于这里讨论的目的,一个指针只是建立了一个间接层次,而忽略了指针值产生的分配问题(这些在第7章讨论过)。

1) 结构和指针引用的三地址码 首先考虑用于域地址计算的三地址码:为了计算 $x.j$ 的地址并存入临时变量 $t1$,使用如下的三地址指令:

```
t1 = &x + field_offset (x,j)
```

如C语句的一个域赋值表达式:

```
x.j = x.i ;
```

能被翻译成如下的三地址码:

```
t1 = &x + field_offset (x,j)
t2 = &x + field_offset (x,i)
*t1 = *t2
```

现在考虑指针。例如,假设 x 被定义成整型指针,例如C声明:

```
int * x;
```

再进一步假设 i 是一个普通整型变量,C赋值语句:

```
*x = i;
```

能被翻译成三地址指令:

```
*x = i
```

赋值语句

```
i = *x;
```

翻译成三地址指令

```
i=*x
```

为了看清楚指针的间接手段是怎样同域地址计算相互作用的,可考虑下面的树结构例子,C变量声明如下:

```
typedef struct treeNode
{ int val;
  struct treeNode * lchild, * rchild;
} TreeNode;
...
TreeNode *p;
```

现在考虑两个典型的赋值

```
p -> lchild = p;
p = p -> rchild;
```

这些语句翻译成三地址码如下：

```
t1 = p + field_offset ( *p, lchild )
*t1 = p
t2 = p + field_offset ( *p, rchild )
p = *t2
```

2) 结构和指针引用的P-代码 给定本次讨论开始时的x定义，x.j的直接地址计算被翻译成下面P-代码

```
lda x
lod field_offset (x,j)
ixa 1
```

赋值语句

```
x.j=x.i;
```

能被翻译成下面的P-代码

```
lda x
lod field_offset (x,j)
ixa 1
lda x
ind field_offset (x,i)
sto
```

注意ind指令在没有计算出x.i的完全地址时是怎样被用来获取它的值的。

在指针情形中(x被定义为int *)。赋值

```
*x = i;
```

翻译成P-代码如下

```
lod x
lod i
sto
```

赋值

```
i = *x;
```

翻译成P-代码如下

```
lda i
lod x
ind 0
sto
```

我们用P-代码可得出赋值

```
p -> lchild = p;
p = p -> rchild;
```

的推断(参见前面p的声明)。这些可翻译成如下P-代码：

```
lod p
lod field_offset ( *p, lchild )
ixa 1
lod p
sto
```

```
lda p
lod p
ind field_offset ( *p, rchild )
sto
```

我们将生成这些三地址码或P-代码的代码生成过程细节留作练习。

8.4 控制语句和逻辑表达式的代码生成

在这一节中，我们描述控制语句不同形式的代码生成。这里主要的部分是结构化的 if 语句和 while 语句，这一节的开头将说明它。这个描述也包括了对 break 语句的说明，但是因为这种语句能简单地用中间代码或目标代码直接实现，所以不讨论低级控制（如 goto 语句）。结构化控制的其他形式，如 repeat 语句（或 do-while 语句）、for 语句和 case 语句（或 switch 语句）的描述将留作练习。在 switch 语句中用到的额外的实现技术称作转移表（jump table），也在练习中描述。

控制语句的中间代码生成——无论是在三地址码还是 P-代码中——都涉及到了标号的产生，这种方式与三地址码中临时变量名的生成相类似，但标号在目标代码中代表要转移的地址。假如在目标代码生成中消除了标号，则在转移中将引发一个代码定位问题，这将在本节的第 2 部分讨论。逻辑表达式或布尔表达式被用作控制测试，但也可以独立地用作数据，这将在接下来的一部分中讨论。尤其是在短循环求值中，它们不同于算术表达式。

最后，在这节中，我们对 if 和 while 语句给出了一个 P-代码生成过程的例子。

8.4.1 if 和 while 语句的代码生成

考虑下面两种 if 和 while 语句形式，这在很多不同的语言中都是相似的（现在给出的是类 C 语法）：

```
if-stmt    if ( exp ) stmt | if ( exp ) stmt else stmt
while-stmt while ( exp ) stmt
```

对这样语句的代码生成的主要问题是：将结构化的控制特性翻译成涉及转移的非结构化等价物，它能被直接实现。编译器以一种标准次序来安排这种语句的代码生成，这种次序有可能高效地使用转移子集，这种转移子集是目标系统所允许的。图 8-1 和图 8-2 是对每一个这种语句的典型代码排列（图 8-1 显示一个 else 部分（虚假的情况），但根据刚刚给定的语法规则这是一个可选项，这个排列对于缺少的 else 部分的修改是很容易的），在这些排列中，仅有两种转移——无条件转移和条件为虚假的转移。条件为真时是不需要转移的“失败”情况。这减少了编译器要产生的转移的数量，这也意味着在中间代码中仅需要两个转移指令。虚假转移已经在程序清单 8-2 中的三地址码中出现过了（如 if-false..goto 语句），此外也在程序清单 8-6 的 P-代码中的三地址码中出现过了（如 fjp 指令）。剩下要介绍的无条件转移，它将在三地址码中用 goto 指令实现，在 P-代码中用 ujp（无条件转移）实现。

1) 控制语句的三地址码 我们假设代码生成器产生了一系列标号 L1、L2...，对于语句

```
if ( E ) S1 else S2
```

生成下面的代码模式：

```
<code to evaluate E to t1>
if_false t1 goto L1
<code for S1>
```

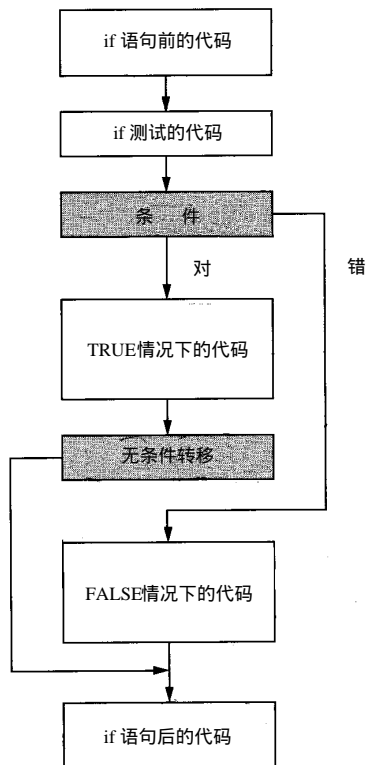


图8-1 if 语句的典型代码排列

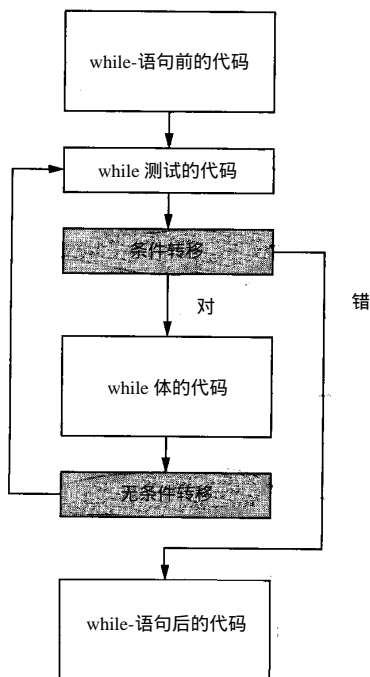


图8-2 while语句的典型代码排列

```

goto L2
label L1
<code for S2>
label L2

```

类似地，while语句

```
while ( E ) S
```

将导致生成下面的三元地址码样式

```

label L1
<code to evaluate E to t1>
if_false t1 goto L2
<code for S>
goto L1
label L2

```

2) 控制语句的P-代码 对于语句

```
if ( E ) S1 else S2
```

生成下面的P-代码样式

```

<code to evaluate E>
fjp L1
<code for S1>
ujp L2

```

```
lab L1
<code for S2>
lab L2
```

对于语句

```
while ( E ) S
```

生成下面的P-代码样式

```
lab L1
<code to evaluate E>
fjp L2
<code for S>
ujp L1
lab L2
```

注意，所有这些代码序列(三地址码和P-代码)都以标号定义结束。我们称这个标号为控制语句的出口标号(exit label)。许多语言提供一个语言结构允许循环在循环体的任意位置退出。例如，C语言提供了**break**语句(这也能被用在switch语句中)。在这些语言中，出口标号对于所有代码生成例程都是可用的，这些代码生成例程在循环体内可以使用。因此如果遇上一个退出语句如break，就产生一个到出口标号的转移。在代码生成期间，将出口标号放入继承属性，这必须被存入栈或作为一个合适的代码生成例程的参教，关于这方面的更多细节，将在这一节的后面给出。

8.4.2 标号的生成和回填

在目标代码生成期间，引起问题的控制语句代码生成的一个特性实际上是对标号的转移必须在标号本身定义之前生成。在中间代码的生成期间这不会有什么问题。由于因向前转移而需要一个标号时，代码生成例程能只是调用标号生成过程，并且一直保存这个标号名字(局部的或在栈中)到知道这个标号的位置为止。假如在目标代码生成期间要产生汇编代码，标号能只是传给汇编器。但如果要生成实际的可执行代码，这些标号就必须被解析成绝对的或相对的代码位置。

产生这样一个向前转移的一个标准方法是：在转移发生的代码处留下空位，或者产生一个虚假的转移(针对虚假地址)。然后当知道实际的转移位置时，这个位置用来回填(backpatch)缺少的代码。这也要求产生代码并保存在内存缓冲区内以易于能频繁地进行回填，或者将代码写入一个临时文件，在需要的时候再重新输入或回填。另一种情况是：回填可能需要在栈中缓冲或者在递归过程中局部地保持。

在回填处理中，更深入的问题出现了，这是由于许多结构中有两种转移，一个短转移(含有128字节)，一个需要更多的代码空间的长转移。在这种情况下，代码生成器可能在短转移时需要插入nop指令或者采取办法缩短代码。

8.4.3 逻辑表达式的代码生成

到目前为止，我们还未提到过逻辑或布尔表达式的代码生成，它们经常作为控制语句的测试而使用。假如中间代码有一个布尔数据类型和逻辑操作符，如**and**和**or**，那么就能在中间代码中计算布尔表达式的值，这与算术表达式一样。它是一个P-代码的例子，能类似地设计出中间代码。然而即使是这个例子，因为大多数系统没有内置的布尔值，所以到目标代码翻译仍需

要将布尔值算术地表示出来。做这个的标准方法是将 **true** 作为1, **false** 作为0, 然后标准的位运算符 **and** 和 **or** 能在大多数系统上用来计算布尔表达式的值。这需要将比较操作符如 **<** 的结果规格化为0或1。在一些系统中, 因为比较操作符本身仅仅设置条件码, 所以需要显式地装入0或1。在那个例子中, 条件转移需要装入合适的值。

如果逻辑操作符是短路(short circuit)的, 更深入地使用转移是必须的。如果不再求它的第2个参数, 那么这个逻辑操作符就被短路了。例如, 假如 *a* 是一个布尔表达式, 值是 **false**, 那么就能立即断定布尔表达式 *a and b* 为假, 也就不去求 *b* 了。类似地, 如 *a* 为真, 那么立刻就能判断出 *a or b* 为真, 也不用求 *b* 了。短路的操作符对代码来说是非常有用的。因为如果操作符没被短路的话, 对第2个表达式的求值可能会引起错误, 例如, 在 C 中这是很平常的:

```
if ((p!=NULL) && ( p->val==0) ) ...
```

这里当 *p* 为可空时 *p->val* 的求值将引起内存错误。

除了它们还返回值以外, 短路布尔操作符类似于 if 语句, 它们经常用 if 表达式定义, 如

```
a and b   if a then b else false
```

和

```
a or b    if a then true else b
```

为产生确保第2个子表达式仅在必要的时候进行求值的代码, 我们必须如同在 if 语句中一样, 使用转移。例如, 对于(表达式 *(x!=0)&&(y==x)*) 的短路 P-代码如下:

```
lod x
ldc 0
neq
fjp L1
lod y
lod x
equ
ujp L2
lab L1
lod FALSE
lab L2
```

8.4.4 if和while语句的代码生成过程样例

这一节用如下简化的文法展示一个控制语句的代码生成过程。

```
stmt    if-stmt | while-stmt | break | other
if-stmt  if( exp ) stmt | if( exp ) stmt else stmt
while-stmt while( exp ) stmt
exp     true | false
```

出于简化目的, 这个语法用 **other** 代表没有包括在文法中的语句(如赋值语句), 它也仅包括常量布尔表达式 **true** 和 **false**。为了显示怎样将 **break** 语句作为参数传递的继承的标号来实现, 它包括了一个 **break** 语句。

下面的 C 声明, 能被用来为这个文法实现一棵抽象语法树

```
typedef enum { ExpKind, IfKind,
               WhileKind, BreakKind, OtherKind } NodeKind;
```

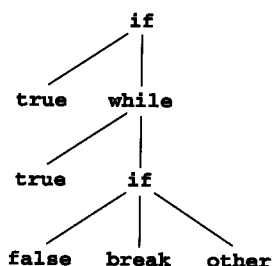


```
typedef struct streenode
{
    NodeKind kind;
    struct streenode * child[3];
    int val; /* used with ExpKind */
} STreeNode;
typedef STreeNode * SyntaxTree;
```

在这棵语法树结构中，1个节点可以有3个孩子(1个有else部分的if节点)。表达式节点包含了真假值(在val域中作为1或0存储)，例如语句

```
if (true) while (true) if (false) break else other
```

有如下的语法树：



这里我们仅显示了每个节点的非空子树^①。

用被给定的typedef和相应的语法树结构，一个产生P-代码的代码生成过程在程序清单8-10中给出，我们对这段代码作了如下注释：

首先，这段代码假设已存在一个emitCode过程(这个过程只是将传给它的字符串打印出来)这段代码也假设返回顺序标号名(如L1、L2、L3...)的无参过程genLabel已存在。

genCode过程有一个额外的标号参数，在为break语句而生成的转移语句中会用到它。仅在处理while语句循环体的递归调用中会改变这个参数。因此，break语句将总是跳出最近的那个嵌套while语句(对genCode的初始调用能用一个空串作为标号参数，任何在while语句外的break语句将产生一个到空标号的转移，这将引发错误)。

请注意标号变量Lab1和Lab2怎样用来在转移和(或)定义仍未决定时保存标号名的。

最后，由于other语句没有对应的实际代码，这个过程只是产生了非P-代码指令“Other”。

程序清单8-10 控制语句的代码生成过程

```
void genCode( SyntaxTree t, char * label)
{
    char codestr[CODESIZE];
    char * lab1, * lab2;
    if (t != NULL) switch (t->kind)
    {
        case ExpKind:
            if (t->val==0) emitCode("ldc false");
            else emitCode("ldc true");
            break;
        case IfKind:
            genCode(t->child[0],label);
            lab1 = genLabel();
            sprintf(codestr,"%s %s","fjp",lab1);
```

① 在这个语法中，标准的“最近嵌套”规则解决了悬挂的二义性，如语法图所示。

```

    emitCode(codestr);
    genCode(t->child[1],label);
    if (t->child[2] != NULL)
    { lab2 = genLabel();
      sprintf(codestr,"%s %s","ujp",lab2);
      emitCode(codestr);}
    sprintf(codestr,"%s %s","lab",lab1);
    emitCode(codestr);
    if (t->child[2] != NULL)
    { genCode(t->child[2],label);
      sprintf(codestr,"%s %s","lab",lab2);
      emitCode(codestr);}
    break;
case WhileKind:
    lab1 = genLabel();
    sprintf(codestr,"%s %s","lab",lab1);
    emitCode(codestr);
    genCode(t->child[0],label);
    lab2 = genLabel();
    sprintf(codestr,"%s %s","fjp",lab2);
    emitCode(codestr);
    genCode(t->child[1],lab2);
    sprintf(codestr,"%s %s","ujp",lab1);
    emitCode(codestr);
    sprintf(codestr,"%s %s","lab",lab2);
    emitCode(codestr);
    break;
case BreakKind:
    sprintf(codestr,"%s %s","ujp",label);
    emitCode(codestr);
    break;
case OtherKind:
    emitCode("Other");
    break;
default:
    emitCode("Error");
    break;
}
}

```

请读者跟踪程序清单 8-10 过程中的操作，并将这条语句的操作显示出来。

```
if (true) while (true) if (false) break else other
```

它产生了如下的代码序列

```

ldc true
fjp L1
lab L2
ldc true
fjp L3
ldc false
fjp L4
ujp L3
ujp L5
lab L4

```

```
Other
lab L5
ujp L2
lab L3
lab L1
```

8.5 过程和函数调用的代码生成

在本章中，过程和函数调用是在一般术语中讨论的最后的语言机制，对这个机制的中间代码和目标代码描述的复杂程度甚至超过了其他语言机制，因为不同的目标机器用相当不同的机制来执行调用，而且调用很大程度上依赖于运行时环境的组织。因此，实现一个对任何目标系统和运行时环境都够用的中间代码表示是困难的。

8.5.1 过程和函数的中间代码

函数调用的中间代码表示的要求可明确地表述如下：首先，有两个实际的机制需要说明——函数过程的定义(definition) (也叫声明(declaration)) 和函数过程的调用(call)^①。定义产生了函数名、参数和代码，但函数却不在那点执行。调用产生了参数的实际值，并且执行一个到函数代码的转移。函数代码被执行和返回。当产生函数代码时，除了它的一般结构，执行的运行时环境还不知道其他情况。这个运行时环境部分由调用者建造，部分也由被调用函数代码建造。任务的分隔是调用次序(calling sequence)的一部分，这在第7章已研究过了。

定义的中间代码必须包括一条标志开始的指令，或称函数代码的入口点(entry point)，还要包括一条标志结束的指令，称为函数返回点(return point)。写出概要如下：

```
Entry instruction
<code for the function body>
Return instruction
```

相似地，函数调用必须有一条指令指示参数计算的开始(为调用而准备的)，然后实际调用指令指示已经构成了参数，到函数代码的实际转移可以发生了。

```
Begin-argument-computation instruction
<code to compute the arguments>
Call instruction
```

不同的中间代码版本在括号括起中的指令的实现上是不相同的。关于环境的信息数量、参数尤为如此，作为每一条指令一部分的函数本身的指令更是这样。这些信息的典型例子包括数字尺寸和参数的定位、栈的大小、局部以及临时变量空间的大小和被调用函数使用寄存器的指示。通常地，将产生一些在指令本身中已包含少量信息的中间代码，而任何必要的信息可以单独地放在过程的符号表中。

1) 过程和函数的三地址码 在三地址码中，入口指令需要给出一个过程入口点的名字，这类似于label指令，因此，这是一条地址指令。我们将其简单地称作 entry，类似地把返

① 通过这段文字，我们发现函数和过程本质上都代表了相同的机制，如没有特殊的说明，在这一节中认为它们是一样的。当然，仅有的区别是：当函数退出时，返回调用者可利用的返回值，并且调用者必须知道在哪儿能发现它。

回指令叫作 **return**。这个指令也是一条地址指令，假如有返回值，那么它必须给出返回值的名字。

例如，考虑C函数定义

```
int f ( int x, int y )
{ return x + y + 1; }
```

这样翻译成如下的三地址码：

```
entry f
t1 = x + y
t2 = t1 + 1
return t2
```

在调用的情况下，实际需要 3 个不同的三地址指令：一个标志参数计算的起点，称作 **begin_args** (是一个零地址指令)；一个被用于重复地说明参数值的名字的指令，我们称其为 **arg** (它必须包括参数值的地址或名字)；最后是实际调用指令，简单地写成 **call**，它也是一个一地址指令(被调用函数的名字或入口点必须给出)。

例如，假如上例中的函数 **f** 已经在C中定义，那么，这个调用

```
f ( 2+3, 4 )
```

翻译成三地址码如下：

```
begin_args
t1 = 2 + 3
arg t1
arg 4
call f
```

在这里按从左到右顺序列出参数。这个顺序可以是不同的(参看7.3.1节)。

2) 过程和函数的P-代码 P-代码的入口指令是 **ent**，返回指令是 **ret**，前面C函数 **f** 的定义可以翻译成P-代码如下：

```
ent f
lod x
lod y
adi
ldc 1
adi
ret
```

注意，**ret**指令不需要一个参数用来指示返回的值。在返回时，返回值被认为已在 P 机器栈顶上。

用于调用的P-代码指令是 **mst** 指令和 **cup** 指令。**mst** 指令表示“标志栈”对应于三地址码指令中的 **begin_args**。它之所以称作“标志栈”的原因是从这种指令生成的目标代码将为一个新调用在栈上建立一个活动记录，这是调用序列中前几个步骤。这通常意味着，对于参数这样的元素，必须在栈中为它分配或“标志”空间。P-代码指令 **cup** 是“调用用户过程”指令，它直接对应于三地址码中的 **call** 指令。取这个名字的原因是 P-代码区分两种调用——**cup** 和 **csp** (调用标准过程)。标准过程是语言定义所需的内部过程，如 pascal 中的 **sin** 和 **abs** 过程(C无内部过程可言)内部过程能用关于它们的操作的特殊知识来提高调用的效率(或者甚至消除这个调用)。这里就不再进一步考虑 **csp** 操作了。

注意，并没有引入等效于三地址码 `arg` 指令的 P-代码指令。相反地，在遇到 `cup` 指令时，将所有参数值都假想成出现在栈顶（按适当的顺序）。这导致了与三地址码稍微不同的调用序列顺序（见练习）。

前面所述的函数 `f`，它的 C 调用例子 `f(2+3,4)` 可以翻译成如下的 P-代码：

```
mst
ldc 2
ldc 3
adi
ldc 4
cup f
```

（再次从左向右计算参数）。

8.5.2 函数定义和调用的代码生成过程

和前面几节一样，我们想展示函数定义和调用语法样例的代码生成过程。要用的文法如下：

```
program    decl-list exp
decl-list  decl-list decl | ε
decl       fn id ( param-list ) = exp
param-list param-list, id | id
exp        exp + exp | call | num | id
call       id ( arg-list )
arg-list   arg-list, exp | exp
```

这个文法定义了一个程序，该程序是函数定义序列，这个序列后跟随着单个表达式。在这个文法中没有变量或赋值，只有参数、函数和可以包括函数调用的表达式。所有值都是整型的，所有函数返回整型，所有函数至少有 1 个参数。这里只有 1 个数字操作（除函数调用外）：整数加法。由这个文法定义的程序例子如下所示：

```
fn f(x)=2+x
fn g(x,y)=f(x)+y
g(3,4)
```

这个程序包含两个函数定义，其后跟着函数 `g` 的调用表达式。在 `g` 中有一个对函数 `f` 的调用。我们想对这个文法定义一个语法树结构，通过下面的 C 声明来进行：

```
typedef enum
{ PrgK, FnK, ParamK, PlusK, CallK, ConstK, IdK }
NodeKind;
typedef struct streenode
{ NodeKind kind;
  struct streenode *lchild, *rchild,
                  *sibling;
  char * name; /* used with FnK, ParamK, CallK, IdK */
  int val; /* used with ConstK */
} StreeNode;
typedef StreeNode * SyntaxTree;
```

在这个树结构里有 7 个不同种类的节点。每个语法树都有一个根节点 `PrgK`。这个节点仅用来将声明和程序表达式绑在一起。一棵语法树只包含一个这样的节点。这个节点的左子树是 `FnK` 节

点的兄弟链表。右子树是相关联的程序表达式。每一个 **FnK** 节点都有一个左子树，它是 **ParamK** 节点的兄弟链表。这些节点定义了参数的名字。每个函数体都是 **FnK** 节点的右子树。除了有一个 **CallK** 节点之外，表达式节点和通常的一样。这个节点包含了所谓函数的名字，并且有一个右子树是参数表达式的兄弟链表。例如，前面样例程序的语法树如图 8-3 所示。为使之表达清楚，在该图中包括了每个节点的节点种类，还有所有的名字 / 值属性。孩子和兄弟用方向区别(兄弟向右，孩子向下)。

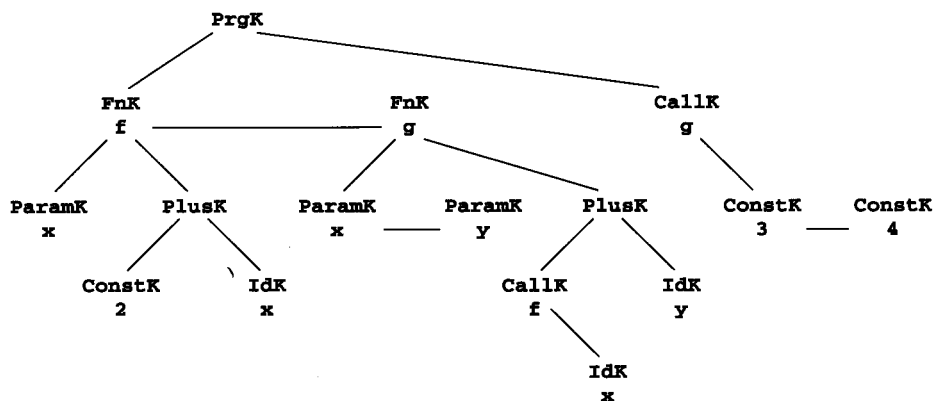


图8-3 上面程序例子的语法树

给定这棵语法树结构，产生 P-代码的代码生成过程在程序清单 8-11 中给出。对这段代码作如下的注释：首先，**PrgK** 节点的代码简单地向树的其余部分递归。**Idk**、**ConstK** 或 **PlusK** 的代码实际上和前面例子中的一样。

程序清单 8-11 函数定义和调用的代码生成过程

```

void genCode( SyntaxTree t)
{ char codestr[CODESIZE];
  SyntaxTree p;
  if (t != NULL)
  switch (t->kind)
  { case PrgK:
    p = t->lchild;
    while (p != NULL)
    { genCode(p);
      p = p->sibling; }
    genCode(t->rchild);
    break;
  case FnK:
    sprintf(codestr,"%s %s","ent",t->name);
    emitCode(codestr);
    genCode(t->rchild);
    emitCode("ret");
    break;
  case ParamK: /* no actions */
    break;
  case ConstK:
    sprintf(codestr,"%s %d","ldc",t->val);
    emitCode(codestr);
  }
}

```

```

        break;
    case PlusK:
        genCode(t->lchild);
        genCode(t->rchild);
        emitCode("adi");
        break;
    case IdK:
        sprintf(codestr, "%s %s", "lod", t->name);
        emitCode(codestr);
        break;
    case CallK:
        emitCode("mst");
        p = t->rchild;
        while (p!=NULL)
        { genCode(p);
          p = p->sibling; }
        sprintf(codestr, "%s %s", "cup", t->name);
        emitCode(codestr);
        break;
    default:
        emitCode("Error");
        break;
}
}

```

这里剩下了FnK、ParamK和CallK的情况。FnK节点的代码只用ent和ret将函数体(右子树)的代码括上了。函数参数没有被访问。实际上参数节点不引起代码的产生,参数已经由调用者产生^①。这也解释了在程序清单8-11中在ParamK节点上为什么没有动作。实际上由于FnK代码的行为,在树的遍历中,不会触及到任何一个ParamK节点。因此这种情况实际上不用讨论。

最后的例子是CallK。代码先发出一个mst指令,然后为每一个参数产生代码,最后发出cup指令。

请读者显示图8-11中的代码生成过程是怎样生成如下的P-代码。给定的程序语法图如图8-3。

```

ent f
ldc 2
lod x
adi
ret
ent g
mst
lod x
cup f
lod y
adi
ret
mst

```

① 因为参数节点参数在活动记录中的相对位置和位移,所以它们有着重要的簿记任务。假设这一点是在其他地方处理的。


```
ldc 3
ldc 4
cup g
```

8.6 商用编译器中的代码生成：两个案例研究

这一节检查两个不同商业编译器(不同的处理器)产生的汇编代码输出。第1个是Borland公司对80×86处理器的3.0版C编译器。第2个是Sun公司对于SparcStation的2.0版C编译器，我们将示出这些编译器对一些代码例子的汇编输出，那些代码例子和用来阐明三地址码和P-代码所用的例子一样[⊖]。这将对代码生成技术和中间代码到目标代码的转化进行更深入的考察，同时它也提供了与TINY编译器生成的机器代码之间有用的比较，TINY的机器代码将在后面的章节中被讨论。

8.6.1 对于80×86的Borland 3.0版C编译器

我们用8.2.1节中使用的赋值表达式开始关于这个编译器输出的例子。这个赋值式是：

```
(x=x+3)+4
```

假设将变量x存于栈中。

对于这个表达式，Borland 3.0 C编译器产生的Intel 80×86上的汇编代码如下：

```
mov     ax, word ptr [bp-2]
add     ax, 3
mov     word ptr [bp-2], ax
add     ax, 4
```

在这段代码中，累加寄存器ax在计算中用作主要的临时变量位置。局部变量x的位置是bp-2，它反应了寄存器bp(基指针)用作框架指针和整型变量在机器中占两个字节。

第1条指令把x的值移到ax中(地址[bp-2]的括号表示一个间接装入而不是一个直接装入)，第2个指令将常量3增加进这个寄存器。第3个指令将这个值移到x的位置。最后，第4条指令将4加入ax，以致于将表达式最后的值留在了寄存器。它可以为进一步的计算所使用。

注意，在第3个指令中赋值用的x的地址不需要预先计算(如P-代码指令lda)。中间代码的静态模拟连同可利用的地址模式的知识能将x的地址计算推迟到要用的时候。

1) 数组引用 用C表达式

```
(a[i+1]=2)+a[j]
```

(参见“数组引用的代码生成过程”部分中的中间代码生成的示例。)作为例子，假设i、j和a被作局部变量声明：

```
int i,j;
int a[10];
```

Borland C编译器为这个表达式产生如下的汇编代码(为使引用变得容易，我们对代码进行了编号)

```
(1)  mov     bx, word ptr [bp-2]
(2)  shl     bx, 1
(3)  lea     ax, word ptr [bp-22]
```

[⊖] 出于这个目的，不考虑编译器的优化。

```

(4)   add    bx,ax
(5)   mov    ax,2
(6)   mov    word ptr [bx],ax
(7)   mov    bx,word ptr [bp-4]
(8)   shl    bx,1
(9)   lea    dx,word ptr [bp-24]
(10)  add    bx,dx
(11)  add    ax,word ptr [bx]

```

因为在这个系统中整数的大小是2字节。 $bp-2$ 是 i 在局部活动记录中的位置。 $bp-4$ 是 j 的位置， a 的基地址是 $bp-24$ ($24 = \text{数组索引尺寸} \times \text{整型尺寸的2个字节} + i \text{和} j \text{的4个字节}$)，因此指令1将 i 的值装入 bx ，指令2将这个值乘以2(左移一位)。指令3将 a 的基地址装入 ax (lea 表示装入有效地址)，这个基地址由于下标表达式 $i+1$ 中的常量1的缘故，已经增加了2个字节。换句话说，编译器已经实现了这样一个代数事实：

$$\begin{aligned}
 address(a[i+1]) &= base_address(a) + (i+1) * elem_size(a) \\
 &= (base_address(a) + elem_size(a)) + i * elem_size(a)
 \end{aligned}$$

指令4计算 $a[i+1]$ 的结果地址，并将其放入 bx 中。指令5将常数2移入 ax 中，指令6将其存入 $a[i+1]$ 的地址中，指令7将 j 值装入 bx ，指令8将这个值乘2，指令9将 a 的基地址装到 dx ，指令10计算 $a[j]$ 地址并放入 bx ，指令11把这个地址中的内容加入到 ax 中。表达式的结果值留在 ax 中。

2) 指针和域引用 假设上一个例子的声明为：

```

typedef struct rec
{
    int i;
    char c;
    int j;
} Rec;

typedef struct treeNode
{
    int val;
    struct treeNode * lchild, * rchild;
} TreeNode;

...
Rec x;
TreeNode *p;

```

同时也假设 x 和 p 被声明为局部变量，也进行了适当的指针分配。

首先考虑涉及的数据类型的尺寸。在 80×86 系统中，整型变量占2个字节，字符变量占1字节，“近”指针占2个字节^①。因此变量 x 有5个字节，变量 p 有2个字节。作为仅有的变量，它们被声明是局部的， x 在活动记录中分配在 $bp-6$ 位置(不能将局部变量分配在偶数字节界上，这是一个典型的限制，因此将不用额外的字节)， p 被分配到寄存器 si 中。更进一步，在结构 Rec 中， i 有偏移量0， c 有偏移量2， j 有偏移量3，而在树节点结构中， val 有偏移量0， $lchild$ 有偏移量2， $rchild$ 有偏移量4。

语句

```
x.j = x.i;
```

① 80×86 有一个更一般的指针叫远指针，有4个字节。

的生成代码是：

```
mov    ax,word ptr [bp-6]
mov    word ptr [bp-3],ax
```

第1条指令将 $x.i$ 装入 ax ，第2条将这个值存到 $x.j$ 中。请注意，对 j 的偏移量计算 $(-6+3=-3)$ 由编译器静态执行。

语句

```
p->l child = p;
```

的生成代码是：

```
mov word ptr [si+2], si
```

请注意怎样将间接和偏移量计算放进同一指令中的。

最后，语句

```
p = p->rchild;
```

的生成代码是

```
mov     si, word ptr [si+4]
```

3) **if**和**while**语句 这里将显示由 Borland C 编译器生成的典型控制语句的代码。所用语句是

```
if (x>y) y++;else x--;
```

和

```
while (x<y) y -= x;
```

在两个例子中 x 和 y 都是局部整型变量。

Borland C 编译器为这个 if 语句产生了如下的 80×86 汇编代码。 x 被放在寄存器 bx 中， y 被放在寄存器 dx 中：

```
cmp     bx,dx
jle     short @1@86
inc     dx
jmp     short @1@114
@1@86:
dec     bx
@1@114:
```

这个代码使用了同图 8-1 一样的顺序组织。但是请读者注意到这个代码并不计算表达式 $x < y$ 的实际逻辑值而只是直接使用条件代码。

Borland C 编译器产生的 while 语句代码如下：

```
jmp     short @1@170
@1@142:
sub     dx,bx
@1@170:
cmp     bx,dx
jl      short @1@142
```

以上运用了不同于图 8-2 的顺序组织。由于将这个测试放在了最后，所以一个初始的无条件的转移转就到这个测试。

4) **函数定义和调用** 将使用的是 C 函数定义：

```
int f( int x, int y)
{ return x+y+1}
```

和一个相应的调用

```
f(2+3, 4)
```

(这些在8.5.1节中已用过。)

首先考虑Borland 编译器对调用`f(2+3, 4)`生成的代码：

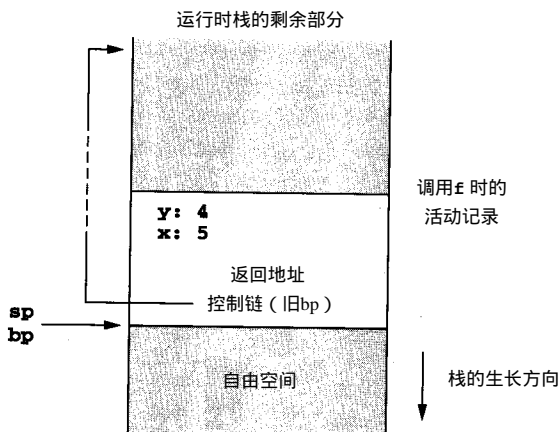
```
mov ax,4
push ax
mov ax,5
push ax
call near ptr _f
pop cx
pop cx
```

注意参数是怎样按相反顺序压入栈的。首先是4，然后再是5(因为 $5=2+3$ 是一个常量表达式，所以编译器预先计算了它的值)。也要注意调用者有责任在调用后从栈中清除参数。这就是为什么在调用后会有两个`pop`指令(目标寄存器`cx`被用作垃圾回收：不再使用弹出的值)的原因。调用本身应该是自解释的：名字`_f`在源名字前有一个下划线，这是C编译器的典型约定，`near ptr`声明表示函数在同一段中(因此仅需2字节的地址)，最后注意，80×86上的`call`指令会自动将返回地址压入栈(由被调用函数执行的对应的`ret`指令将它弹出)。

现在考虑由Borland C编译器的生成的，函数`f`定义语句的代码：

```
_f      proc      near
        push      bp
        mov       bp,sp
        mov       ax,word ptr [bp+4]
        add       ax,word ptr [bp+6]
        inc       ax
        jmp       short @1@58
@1@58:
        pop       bp
        ret
_f      endp
```

这段代码中的许多指令都是一目了然的。由于调用者除了计算和压入参数外没有做活动记录的构造，这段代码必须在函数体代码被执行前完成它的构造，这是第2、第3个指令的任务，它们保存控制链(`bp`)到栈中，然后将`bp`设成现在的`sp`。经过这样操作，栈中内容如下：



栈中的返回值在控制链(旧的bp)和由调用者call指令产生的参数之间。因此,旧的bp在栈顶,返回值在bp+2(在这个例子中地址是2字节的)。参数x在bp+4中,参数y在bp+6。f的函数体对应于如下的代码:

```
mov ax,word ptr [bp+4]
add ax,word ptr [bp+6]
inc ax
```

把x装进ax,再加y到ax中,然后再把ax中内容加1。

最后,代码执行一个转移(尽管这里并不必要,但总也要产生,这是因为在函数中嵌的return语句会需要它),从这个栈中恢复旧的bp,然后再返回到调用者。留在ax中的返回值对调用者来说是可用的。

8.6.2 Sun SparcStation的Sun 2.0 C编译器

我们重复使用前一节中的代码例子说明这个编译器的输出。同样地,以赋值语句

```
(x=x+3)+4
```

开始并且假设表达式中的变量x储存在局部栈框架中。

Sun C编译器产生汇编代码与Borland代码极为相似:

```
ld      [%fp+-0x4],%o1
add     %o1,0x3,%o1
st      %o1,[%fp+-0x4]
ld      [%fp+-0x4],%o2
add     %o2,0x4,%o3
```

在这些代码中,寄存器名以百分号打头,常数以字符0x(x=十六进制)打头。所以,例如0x4就是十六进制4(与十进制相同)。第1条指令把x的值(在位置fp-4,由于整形为4个字节长度)赋给了寄存器o1^①。(注意源位置在左而目的位置在右,与Intel 80x86习惯相反。)第2条指令把常数3加到o1,而第3条把o1存到x的位置。最后,x的值再次装入,这次是到o2寄存器^②,并且再加上了4,结果被放到寄存器o3,这是表达式的最终结果。

1) 数组引用 表达式

```
(a[i+1]=2)+a[j]
```

以及全部局部变量都被Sun编译器翻译成下面的汇编代码(带有指令编号以便引用):

```
(1)  add     %fp,-0x2c,%o1
(2)  ld      [%fp+-0x4],%o2
(3)  sll     %o2,0x2,%o3
(4)  mov     0x2,%o4
(5)  st      %o4,[%o1+%o3]
(6)  add     %fp,-0x30,%o5
(7)  ld      [%fp+-0x8],%o7
(8)  sll     %o7,0x2,%10
(9)  ld      [%o5+%10],%11
```

① 在SparcStation中寄存器由一个小写字母后跟一个数字来表示。不同的字母表示不同的“寄存器窗口”,这会随着上下文的改变而改变并大约对应于寄存器的不同用途。本章除了一个涉及过函数调用(见8.5节)的全部例子外,不同的字母之间的区别都是看不见的。

② 这个步骤没有出现在Borland代码中,并很容易优化掉。参见8.9节。

```
(10)  mov    0x2,%12
(11)  add    %12,%11,%13
```

这段代码执行的计算受此结构中整形为 4 个字节的因素的影响。因此，*i* 的位置是 *fp-4* (在代码中写为 *%fp+-0x4*)，*j* 的位置是 *fp-8* (*%fp+-0x8*)，而 *a* 的基地址是 *fp-48* (由于十进制 48 = 十六进制 30，写成 *%fp+-0x30*)。指令 1 将 *a* 的基地址装入到寄存器 *o1*，此地址已修改为下标 *i+1* 中的常数 1 减去 4 个字节 (如同 Borland 代码) (*2c hex = 44 = 48 - 4*)。指令 2 将 *i* 的值装入到寄存器 *o2* 中。指令 3 把 *o3* 乘以 4 (左移 2 位) 并把结果存放在 *o3* 中。指令 4 把常数 2 装入寄存器 *o4*，而指令 5 将其存入地址 *a[i+1]*。指令 6 计算 *a* 的基地址并存入 *o5*，指令 7 将 *j* 的值装入 *o7*。而指令 8 将其乘以 4，把结果存入寄存器 *10*，最后，指令 9 把 *a[j]* 的值装入 *11*，指令 10 重新将 2 装入 *12*，指令 11 将它们相加，把结果 (也就是表达式的最终结果) 存入寄存器 *13*。

2) 指针和域引用 考虑与前面相同的 Sun 例子 (参见第 8.6.1 节的“指针和域引用”部分)。数据类型大小如下：整形变量占 4 个字节，字符变量占 1 个字节，指针占 4 个字节。其实所有的变量 (包括结构域) 仅分配了 4 字节。这样，变量 *x* 占了 12 字节，变量 *p* 占了 4 字节，而 *i*、*c* 和 *j* 的偏移分别为 0、4 和 8，这同 *val*、*lchild* 和 *rchild* 的偏移一样。编译器在活动记录中定位 *x* 和 *p*：*x* 定位在 *fp-0xc* (hex *c=12*)，*p* 定位在 *fp-0x10* (hex *10=16*)。

为赋值语句

```
x.j = x.i;
```

产生的代码

```
ld    [%fp+-0xc], %o1
st    %o1, [%fp+-0x4]
```

这段代码把 *x.i* 的值装入寄存器 *o1*，然后再存储到 *x.j* (请注意，*x.j = -12+8=4* 的偏移是静态计算的)。

指针赋值

```
p->lchild=p;
```

的目标代码为

```
ld [%fp+-0x10], %o2
ld [%fp+-0x10], %o3
st %o3, [%o2+0x4]
```

在这里 *p* 的值装载到寄存器 *o2* 和 *o3*，其中的一个拷贝 (*o2*) 用作存储另一个拷贝到 *p->lchild* 位置的基地址。最后，赋值语句

```
p = p->rchild;
```

的目标代码

```
ld [%fp+-0x10], %o4
ld [%o4 +0x8], %o5
st %o5, [%fp+-0x10]
```

在这段代码中 *p* 的值装入寄存器 *o4*，然后作为基地址以装入 *p->rchild* 的值。最后一条指令将这个值存入 *p* 的位置。

3) If 和 While 语句 我们如同 Borland 编译器一样展示 Sun SparcStation C 编译器为同一段典型控制语句产生的代码

```
if (x>y) y++; else x--;
```

和

```
while (x<y) y -= x;
```

x和**y**为局部整形变量。

Sun SparcStation编译器为if语句产生下面代码，**x**和**y**定位在局部动态记录中偏移为-4和-8：

```
ld      [%fp+-0x4],%o2
ld      [%fp+-0x8],%o3
cmp     %o2,%o3
bg      L16
nop
b       L15
nop
L16:
ld      [%fp+-0x8],%o4
add     %o4,0x1,%o4
st      %o4,[ %fp+-0x8]
b       L17
nop
L15:
ld      [%fp+-0x4],%o5
sub     %o5,0x1,%o5
st      %o5,[ %fp+-0x4]
L17:
```

这里的nop指令之所以跟随在每个分支之后是因为 Sparc是流水线的（分支延迟且下一条指令总是在分支生效前执行）。请注意后续的组织与图8-10相反，真状态放在虚假的状态之后。

Sun编译器为while循环产生的代码为

```
ld      [%fp+-0x4],%o7
ld      [%fp+-0x8],%i10
cmp     %o7,%i10
bl      L21
nop
b       L20
nop
L21:
L18:
ld      [%fp+-0x4],%i11
ld      [%fp+-0x8],%i12
sub     %i12,%i11,%i12
st      %i12,[ %fp+-0x8]
ld      [%fp+-0x4],%i13
ld      [%fp+-0x8],%i14
cmp     %i13,%i14
bl      L18
nop
b       L22
nop
L22:
L20:
```

除了测试代码也在开始处复制之外，代码使用类似 Borland编译器的安排。此外在代码结尾处，

“什么也不做”分支(到标号L22)会被优化步骤轻易删除。

4) 函数定义与调用 我们使用与前面相同的定义

```
int f ( int x, int y )
{ return x+y+1; }
```

和同样的调用

```
f ( 2+3, 4 )
```

Sun编译器产生的调用代码为：

```
mov      0x5, %o0
mov      0x4, %o1
call     _f, 2
```

为定义f产生的代码为

```
_f:
    !#PROLOGUE# 0
    sethi    %hi(LF62),%g1
    add      %g1,%lo(LF62),%g1
    save     %sp,%g1,%sp
    !#PROLOGUE# 1
    st       %i0,[%fp+0x44]
    st       %i1, [%fp+0x48]
L64:
    .seg     "text"
    ld       [%fp+0x44],%o0
    ld       [%fp+0x48],%o1
    add      %o0,%o1,%o0
    add      %o0,0x1,%o0
    b        LE62
    nop
LF62:
    mov      %o0,%i0
    ret
    restore
```

我们将不详细讨论这段代码，而只给出以下的说明：首先，调用通过寄存器 o0和o1而不是用栈来传递参数。调用使用数字 2指出用于这个目的寄存器数。调用还执行了一些簿记功能，这些都不再深入讨论，除了指出调用之后“o”寄存器(如o1和o2)变成“i”寄存器(例如i0和i1)(调用完成后“i”寄存器又变回“o”寄存器)⊖。

定义f的代码也以一些完成调用序列的簿记指令(在!#PROLOGUE#注释之间)开始，这些也不进一步讨论了。代码然后将参数值存入栈中(对于“i”寄存器，与调用者的“o”寄存器等同)。在返回值计算完之后，将其存入寄存器i0(返回后就可以寄存器o0访问)。

8.7 TM：简单的目标机器

本节之后将展示 TINY语言的代码生成器。为了使这成为有意义的工作，我们产生的目标

⊖ 不规范地说“o”代表“output”，“i”代表“input”，这种寄存器在调用前后的变换由 Sparc结构的寄存器窗口(register window)机制执行。

代码可直接用于易于模拟的简单机器。这个机器称为 TM(Tiny Machine)。附录C提供了TM模拟器的完整的C程序，它可以用来运行 TINY代码生成器产生的代码。本节描述完整的 TM结构和指令集以及附录C中的模拟器。为了便于理解，我们用 C代码片段辅助说明，而TM指令本身总是以汇编代码而不是二进制或十六进制形式给出(模拟器总是只读入汇编代码)。

8.7.1 Tiny Machine的基本结构

TM由只读指令存储区、数据区和 8个通用寄存器构成。它们都使用非负整数地址且以 0开始。寄存器7为程序计数器，它也是唯一的专用寄存器，如下面所描述的那样。C的声明：

```
#define IADDR_SIZE ...
    /* size of instruction memory */
#define DADDR_SIZE...
    /* size of data memory */
#define NO_REGS 8 /* number of registers */
#define PC_REG 7

Instruction iMem[IADDR_SIZE];
int dMem[DADDR_SIZE];
int reg[NO_REGS];
```

将用于描述。

TM执行一个常用的取指-执行循环

```
do
    /* fetch */
    current Instruction = iMem [reg[pcRegNo]++];
    /* execute current instruction */
    ...
while (!(halt||error));
```

在开始点，Tiny Machine将所有寄存器和数据区设为 0，然后将最高正规地址的值(名为 DADDR_SIZE-1)装入到 dMem[0]中。(由于程序可以知道在执行时可用内存的数量，所以它允许将存储器很便利地添加到 TM上)。TM然后开始执行 iMem[0]指令。机器在执行到 HALT指令时停止。可能的错误条件包括 IMEM_ERR(它发生在取指步骤中若 reg[PC_REG]<0或 reg[PC_REG] IADDR_SIZE时)，以及两个条件 DMEM_ERR和 ZERO-DIV，它们发生在下面描述的指令执行中。

程序清单 8-12 给出 TM的指令集以及每条指令效果的简短描述。基本指令格式有两种：寄存器，即 RO指令。寄存器-存储器，即 RM指令。寄存器指令有如下格式：

```
opcode r, s, t
```

程序清单 8-12 Tiny Machine 的完全指令集

RO 指令

格式	<i>opcode r,s,t</i>	
操作码	效果	
HALT	停止执行(忽略操作数)	
IN	reg[r]	从标准读入整形值(<i>s</i> 和 <i>t</i> 忽略)
OUT	reg[r]	标准输出(<i>s</i> 和 <i>t</i> 忽略)

```

ADD      reg[r] = reg[s] + reg[t]
SUB      reg[r] = reg[s] - reg[t]
MUL      reg[r] = reg[s] * reg[t]
DIV      reg[r] = reg[s] / reg[t](可能产生ZERO_DIV)

```

RM 指令

格式 *opcode r,d(s)*

($a = d + \text{reg}[s]$; 任何对 $\text{dmem}[a]$ 的引用在 $a < 0$ 或 $a \geq \text{DADDR_SIZE}$ 时产生 DMEM_ERR)

操作码 效果

```

LD       reg[r] = dMem[a](将a中的值装入r)
LDA      reg[r] = a (将地址a直接装入r)
LDC      reg[r] = d (将常数d直接装入r, 忽略s)
ST       dMem[a] = reg[r](将r的值存入位置a)
JLT      if (reg[t] < 0) reg[PC_REG] = a (如果r小于零转移到a, 以下类似)
JLE      if (reg[t] <= 0) reg[PC_REG] = a
JGE      if (reg[t] > 0) reg[PC_REG] = a
JGT      if (reg[t] > 0) reg[PC_REG] = a
TEQ      if (reg[t] == 0) reg[PC_REG] = a
JNE      if (reg[t] != 0) reg[PC_REG] = a

```

这里操作数 r 、 s 、 t 为正规寄存器(在装入时检测)。这样这种指令有3个地址, 且所有地址都必须为寄存器。所用算术指令被限制到这种格式, 以及两个基本输入/输出指令。

一条寄存器-存储器指令有如下格式:

opcode r,d(s)

在代码中 r 和 s 必须为正规的寄存器(装入时检测), 而 d 为代表偏移的正、负整数。这种指令为两地址指令, 第1个地址总是一个寄存器, 而第2个地址是存储器地址 a , 用 $a = d + \text{reg}[s]$ 给出, 这里 a 必须为正规地址 ($0 \leq a < \text{DADDR_SIZE}$)。如果 a 超出正规的范围, 在执行中就会产生 DMEM_ERR 。

RM指令包括对应于3种地址模式的3种不同的装入指令: “装入常数”(LDC), “装入地址”(LDA)和“装入内存”(LD)。另外, 还有1条存储指令和6条转移指令。

在RD和RM中, 即使其中一些可能被忽略, 所有的3个操作数也都必须表示出来。这是由于简化了装载器, 它仅区分两类指令(RO和RM)而不允许在一类中有不同的指令格式^①。

程序清单8-12和到此为止的TM讨论表示了完整的TM结构。特别需要指出的是: 除了 pc 之外没有特殊寄存器(没有 sp 或 fp), 其他再也没有硬件栈和其他种类的要求。因此, TM的编译器必须完全手工维护运行时环境组织。虽然这看起来有点不切实际, 但它所有操作在需要时必须显式产生的优点。

由于指令集是最小的, 这就需要一些说明来指出它们如何被用来构造大部分标准程序语言操作(实际上, 这个机器如果不去满足少量高级语言的话已经足够了)。

1) 算术运算中目标寄存器、IN以及装入操作先出现, 然后才是源寄存器。这类似于 80×86 而不同于 Sun SparcStation。对于目标和源的寄存器没有限制: 特别地, 目标和源寄存器可以相同。

① 这也使代码生成更容易了, 因为对两类指令只需两种例程。

2) 所有的算术操作都限制在寄存器之上。没有操作 (除了装入和存储操作) 是直接作用于内存的。这一点TM与诸如Sun Sparc Station的RISC机器相似。另一方面, TM只有8个寄存器, 而大部分RISC处理器有至少32个^①。

3) 没有浮点操作和浮点寄存器。为TM增加浮点操作和寄存器的协处理器并不困难。在普通寄存器和内存之间转换浮点数时要小心一些。请参阅练习。

4) 与其他一些汇编代码不同, 这里没有在操作数中指定地址模式的能力 (比如LD#1表示立即模式, 或LD @a表示间接)。作为代替的是对应不同模式的不同指令: LD是间接, LDA是直接, 而LDC是立即。实际上TM只有很少的地址选择。

5) 在指令中没有限制使用pc。实际上由于没有无条件转移指令, 因此必须由将pc作为LDA指令的目标寄存器来模拟:

```
LDA 7,d(s)
```

这条指令效果为转移到位置 $a = d + \text{reg}[s]$ 。

6) 这里也没有间接转移指令, 不过也可以模拟, 如果需要也可以使用LD指令, 例如,

```
LD 7,0(1)
```

转移到由寄存器1指示地址的指令。

7) 条件转移指令(JLT等)可以与程序中当前位置相关, 只要把pc作为第2个寄存器, 例如

```
JEQ 0,4(7)
```

导致TM在寄存器0的值为0时向前转移5条指令, 无条件转移也可以与pc相关, 只要pc两次出现在LDA指令中, 这样,

```
LDA 7,-4(7)
```

执行无条件转移回退3条指令。

8) 没有过程和JSUB指令, 作为代替, 必须写出

```
LD 7,D(s)
```

其效果是转移到过程, 其入口地址为 $\text{dMem}[d+\text{reg}[s]]$ 。当然, 要记住先保存返回地址, 类似于执行

```
LDA 0,1(7)
```

它将当前pc 值加1放到 $\text{reg}[0]$ (那是我们要返回地地方, 假设下一条指令是实际的转移到过程)。

8.7.2 TM模拟器

这个模拟器接受包含上面所述的TM指令的文本文件, 并有以下约定:

- 1) 忽略空行。
- 2) 以星号打头的行被认为是注释而忽略。
- 3) 任何其他行必须包含整数指示位置后跟冒号再接正规指令。任何指令后的文字都被认为是注释而被忽略掉。

TM模拟器没有其他特征了——特别是没有符号标号和宏能力。程序清单 8-13为一个手写的TM程序对应于程序清单8-1的TINY程序。严格地说, 程序清单8-13中代码尾部不需要HALT

^① 由于TM寄存器的数量增加很容易, 因为基本代码生成无需如此, 所以我们不必这样做。见本章最后的练习。

指令，由于TM模拟器在装入程序之前已设置了所有指令位置直到HALT。然而，将其作为一个提醒是很有用的——以及作为转移退出程序的目标。

程序清单8-13 显示约定的程序

```

* This program inputs an integer, computes
* its factorial if it is positive,
* and prints the result
0:   IN    0, 0, 0      r0 = read
1:   JLE   0, 6 (7)     if 0 < r0 then
2:   LDC   1,1,0        r1 = 1
3:   LDC   2, 1, 0      r2=1
                        * repeat
4:   MUL   1, 1, 0      r1 = r1*r0
5:   SUB   0, 0, 2      r0 = r0-r2
6:   JNE   0, -3 (7)    until r0 == 0
7:   OUT   1, 0, 0      write r1
8:   HALT  0, 0, 0      halt
* end of program

```

此外没有必要如程序清单 8-13中那样将位置升序排列。每个输入行足够指出“将这条指令存在这个位置”：如果TM程序打在卡片上，那么在掉到地板上之后再读入还是工作得很完美。TM模拟器的这种特性可能引起阅读程序时的混淆，为了在没有符号标号的情况下反填转移，代码要能不必回翻代码文件就能完成反填。例如，代码生成器可能产生程序清单 8-13代码如下：

```

0: IN    0,0,0
2: LDC   1,1,0
3: LDC   2,1,0
4: MUL   1,1,0
5: SUB   0,0,2
6: JNE   0,-3(7)
7: OUT   1,0,0
1: JLE   0,6(7)
8: HALT  0,0,0

```

因为在知道if语句体后面的位置之前，向前转移的指令1没法生成。

如果程序清单8-13的程序在文件fact.tm中，那么这个文件可以用下面的示例任务装入并执行(如果没有扩展名，TM模拟器自动假设.tm)：

```

tm fact
TM simulation ( enter h for help ) ...
Enter command: g
Enter value for IN instruction: 7
OUT instruction prints: 5040
HALT: 0, 0, 0
Halted
Enter command: q
Simulation done

```

g命令代表“go”，它表示程序用当前pc中的内容(装入之后为0)开始执行到看到HALT指令为止。完整的模拟器命令可以用h命令得到，将打出以下列表：

Commands are:

```
s(step <n>      Execute n ( default 1 ) TM instructions
g(o            Execute TM instructions until HALT
r(egs         print the contents of the registers
i(Mem <b <n>> Print n iMem locations strarting at b
d(Mem <b <n>> Print n dMem locations strarting at b
t(race        Toggle instructions trace
p(rint        Toggle print of total instructions executed ('go' only)
c(lear        Reset simulator for new execution of program
h(elp        Cause this list of commands to printed
q(uit        Terminate the simulation
```

命令中的右括号指示命令字母衍生的记忆法(使用多个字母也可以，但模拟器只检查首字母)。尖括号< >表示可选参数。

8.8 TINY语言的代码生成器

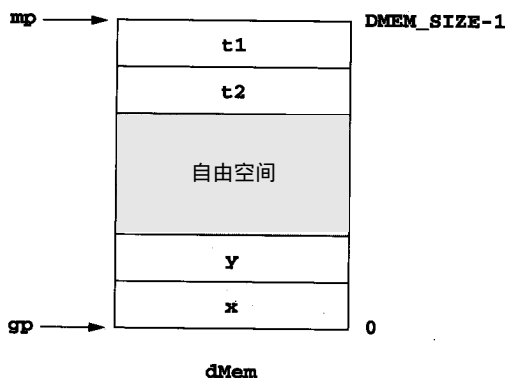
现在要描述TINY语言的代码生成器。我们假设读者熟悉TINY编译器的先前步骤，特别是第3章中描述的分析器产生的语法树结构、第6章描述的符号表构造，以及第7章的运行时环境。

本节首先描述TINY代码生成器和TM的接口以及代码生成必需的实用函数。然后再说明代码生成的步骤。接着，描述TINY编译器和TM模拟器的结合。最后讨论全书使用的示例TINY程序的目标代码。

8.8.1 TINY代码生成器的TM接口

一些代码生成器需要知道的有关TM的信息已封装在文件code.h和code.c中，在附录B中有该程序，分别是第1600行到第1685行和第1700行到第1796行。此外还在文件中放入了代码发行函数。当然，代码生成器还是要知道TM指令的名字，但是这些文件分离了指令格式的详细说明和目标代码文件的位置以及运行时使用特殊寄存器。code.c文件完全可以将指令序列放到特别的iMem位置，而代码生成器就不必追踪细节了。如果TM装载器要改进，也就是说允许符号标号并去掉数字编号，那么将很容易将标号生成和格式变化加入到code.c文件中。

现在我们复习一下code.h文件中的常数和函数定义。首先是寄存器值的定义(1612, 1617, 1623, 1626和1629行)。明显地，代码生成器和代码发行实用程序必须知道pc。另外还有TINY语言的运行时环境，如前一节所述，将数据存储时的顶部分配给临时存储(以栈方式)而底部则分配给变量。由于TINY中没有活动记录(于是也就没有fp)(没有作用域和过程调用)，变量和临时存储的位置可认为是绝对的。然而，TM机的LD操作不允许绝对地址，而必须有一个寄存器基值来计算存储装入的地址。这样我们分配两个寄存器，称为mp(内存指针)和gp(全程指针)来指示存储区的顶部和底部。mp将用于访问临时变量，并总是包含最高正规内存位置，而gp用于所有命名变量访问，并总是包含0。这样由符号表计算的绝对地址可以生成相对gp的偏移来使用。例如，如果程序使用两个变量x和y，并有两个临时值存在内存中，那么dMem将如下所示：



在本图中， $t1$ 的地址为 $0(mp)$ ， $t2$ 为 $-1(mp)$ ， x 的地址为 $0(gp)$ ，而 y 为 $1(gp)$ 。在这个实现中， gp 是寄存器5， mp 是寄存器6。

另两个代码生成器将使用的寄存器是寄存器0和1，称之为“累加器”并命令名为 ac 和 $ac1$ 。它们被当作相等的寄存器来使用。通常计算结果存放在 ac 中。注意寄存器2、3和4没有命名(且从不使用1)。

现在来讨论7个代码发行函数，原型在`code.h`文件中给出。如果`TraceCode`标志置位，`emitComment`函数会以注释格式将其参数串打印到代码文件中的新行中。下两个函数`emitRO`和`emitRM`为标准的代码发行函数用于RO和RM指令类。除了指令串和3个操作数之外，每个函数还带有1个附加串参数，它被加到指令中作为注释(如果`TraceCode`标志置位)。

接下来的3个函数用于产生和反填转移。`emitSkip`函数用于跳过将来要反填的一些位置并返回当前指令位置且保存在`code.c`内部。典型的应用是调用`emitSkip(1)`，它跳过一个位置，这个位置后来会填上转移指令，而`emitSkip(0)`不跳过位置，调用它只是为了得到当前位置以备后来的转移引用。函数`emitBackup`用于设置当前指令位置到先前位置来反填，`emitRestore`用于返回当前指令位置给先前调用`emitBackup`的值。典型地，这些指令在一起使用如下：

```
emitBackup(savedLoc) ;
/* generate backpatched jump instruction here */
emitRestore() ;
```

最后代码发行函数(`emitRM_Abs`)用来产生诸如反填转移或任何由调用`emitSkip`返回的代码位置的转移的代码。它将绝对代码地址转变成 pc 相关地址，这由当前指令位置加1(这是 pc 继续执行的地方)减去传进的位置参数，并且使用 pc 做源寄存器。通常地，这个函数仅用于条件转移，比如`JEQ`或使用`LDA`和 pc 作为目标寄存器产生无条件转移，如前一小节所述的那样。

这样就描述完了TINY代码生成实用程序，我们来看一看TINY代码生成器本身的描述。

8.8.2 TINY代码生成器

TINY代码生成器在文件`cgen.c`中，其中提供给TINY编译器的唯一接口是`CodeGen`，其原型为：

```
void CodeGen (void);
```

在接口文件`cgen.h`中给出了唯一的定义。附录B中有完整的`cgen.c`文件，参见第1900行到第2111行。

函数`CodeGen`本身(第2095行到第2111行)所做的事极少：产生一些注释和指令(称为标准

序言(standard prelude))、设置启动时的运行时环境，然后在语法树上调用 **cGen**，最后产生 **HALT**指令终止程序。标准序言由两条指令组成：第1条将最高正规内存位置装入 **mp**寄存器(**TM**模拟器在开始时置0)。第2条指令清除位置0(由于开始时所有寄存器都为0，**gp**不必置0)。

函数**cGen**(第2070行到第2084行)负责完成遍历并以修改过的顺序产生代码的语法树，回想 **TINY**语法树定义给出的格式：

```
typedef enum { StmtK, ExpK } NodeKind;
typedef enum { IfK, RepeatK, AssignK, ReadK, WriteK } StmtKind;
typedef enum { OpK, ConstK, IdK } ExpKind;

#define MAXCHILDREN 3
typedef struct treeNode
{
    struct treeNode * child[MAXCHILDREN];
    struct treeNode * sibling;
    int linenos;
    NodeKind nodekind;
    union { StmtKind stmt; ExpKind exp; } kind;
    union { TokenType op;
        int val;
        char * name; } attr;
    ExpType type;
} TreeNode;
```

这里有两种树节点：句子节点和表达式节点。如果节点为句子节点，那么它代表 5种不同**TINY**语句(**if**、**repeat**、赋值、**read**或**write**)中的一种，如果节点为表达式节点，则代表 3种表达式(标识符、整形常数或操作符)中的一种。函数**cGen**仅检测节点是句子或表达式节点(或空)，调用相应的函数**genStmt**或**genExp**，然后在同属上递归调用自身(这样同属列表将以从左到右格式产生代码)。

函数**genStmt**(第1924行到第1994行)包含大量**switch**语句来区分5种句子，它产生代码并在每种情况递归调用**cGen**，**genExp**函数(第1997行到第2065行)也与之类似。在任意情况下，子表达式的代码都假设把值存到 **ac**中而可以被后面的代码访问。当需要访问变量时(赋值和**read**语句以及标识符表达式)，通过下面操作访问符号表：

```
loc = lookup(tree->attr.name);
```

loc的值为问题中的变量地址并以 **gp**寄存器基准的偏移装入或存储值。

其他需要访问内存的情况是计算操作符表达式的结果，左边的操作数必须存入临时变量直到右边操作数计算完成。这样操作符表达式的代码包含下列代码生成序列在操作符应用 (第2021行到第2027行)之前：

```
cGen(p1); /* p1 = left child */
emitRM("ST",ac,tmpOffset--,mp,"op: push left");
cGen(p2); /* p2 = right child */
emitRM("LD",ac1,++tmpOffset,mp,"op: load left");
```

这里的**tmpOffset**为静态变量，初始为用作下一个可用临时变量位置对于内存顶部(由**mp**寄存器指出)的偏移。注意**tmpOffset**如何在每次存入后递减和读出后递增。这样 **tmpOffset**可以看成是“临时变量栈”的顶部指针，对 **emitRM**函数的调用与压入和弹出该栈相对应。这在临时变量在内存中时保护它们。在以上代码之前执行实际动作，左边的操作数将在寄存器1(**ac1**)中而右边操作数在寄存器0(**ac**)中。如果是算术操作的话，就产生相应的 **RO**操作。

比较操作符的情况有少许差别。TINY语言的语法(如语法分析器中的实现,参见前面章节)仅在if语句和while语句的测试表达式中允许比较操作符。在这些测试之外也没有布尔变量或值,比较操作符可以在这些语句的代码生成内部处理。然而,这里我们用更通常的方法,它更广泛应用于包含逻辑操作与/或布尔值的语言,并将测试结果表示为0(假)或1(真),如同在C中一样。这要求常数0或1显式地装入ac,用转移到执行正确装载来实现这一点。例如,在小于操作符的情况下,产生了以下代码,代码产生将计算左边操作数存入寄存器1,并计算右边操作数存入寄存器0:

```
SUB    0,1,0
JLT    0,2(7)
LDC    0,0(0)
LDA    7,1(7)
LDC    0,1(0)
```

第1条指令将左边操作数减去右边操作数,结果放入寄存器0,如果<为真,结果应为负值,并且指令JLT 0,2(7)将导致跳过两条指令到最后一条,将值1装入ac,如果<为假,将执行第3条和第4条指令,将0装入ac然后跳过最后一条指令(回忆TM的描述,LDA使用pc为寄存器引起无条件转移)。

我们将以if-语句(第1930行到第1954行)的讨论来结束TINY代码生成器的描述。其余的情况留给读者。

代码生成器为if语句所做的第1个动作是为测试表达式产生代码。如前所述测试代码,在假时将0存入ac,真时将1存入。生成代码接下来要产生一条JEQ到if语句的else部分。然而这些代码的位置当前是未知的,这是因为then部分的代码还要生成。因此,代码生成器用emitSkip来跳过后面的语句并保存位置用于反填:

```
savedLoc1 = emitSkip(1);
```

代码生成继续处理if算语句的then部分。之后必须无条件转移跳过else部分。同样转移位置未知,于是这个转移的位置也要跳过并保存位置:

```
savedLoc2 = emitSkip(1);
```

现在,下一步是产生else部分的代码,于是当前代码位置是正确的假转移的目标,要反填到位置savedLoc1。下面的代码处理之:

```
currentLoc = emitSkip(0);
emitBack up(savedLoc1);
emitRM_Abs("JEQ",ac,currentLoc,"if: jmp to else");
emitRestors();
```

注意emitSkip(0)调用是如何用来获取当前指令位置的,以及emitRM_Abs过程如何用于将绝对地址转移变换成pc相关的转移,这是JEQ指令所需的。之后就可以为else部分产生代码了,然后用类似的代码将绝对转移(LDA)反填到savedLoc2。

8.8.3 用TINY编译器产生和使用TM代码文件

TINY代码生成器可以和谐地与TM模拟器一起工作。当主程序标志NO_PARSE、NO_ANALYZE和NO_CODE都置为假时,编译器创建.tm后缀的代码文件(假设源代码中无错误)并将TM指令以TM模拟器要求的格式写入该文件。例如,为编译并执行sample.tny程序,只要发出下面命令:

```

tiny sample
<listing produced on the standard output>
tm sample
<execution of the tm simulator>

```

为了跟踪的目的，有一个 `TraceCode` 标志在 `globals.h` 中声明，其定义在 `main.c` 中。如果标志为 `TRUE`，代码生成器将产生跟踪代码，在代码文件中表现为注释，指出每条指令或指令序列在代码生成器的何处产生以及产生原因。

8.8.4 TINY编译器生成的TM代码文件示例

为了详细说明代码生成是如何工作的，我们在程序清单 8-14 中展示了 TINY 代码生成器生成的程序清单 8-1 中示例程序的代码，由于 `TraceCode = TRUE`，所以也产生了代码注释。这个代码文件有 42 条指令，其中包括来自标准序言的两条指令。将它与程序清单 8-13 中手写的程序中的漂亮指令对比，我们可以明显看出一些不够高效之处。特别地，程序清单 8-13 的程序高效地使用了寄存器，除了寄存器之外没有再也用到内存。程序清单 8-14 代码正相反，没有使用超过两个寄存器并执行了许多不必要的存储和装入。特别愚蠢的是处理变量值的方法，其装入只是为了再次存储到临时变量中，如下所示：

```

16:      LD      0,1(5) load id value
17:      ST      0,0(6) op: push left
18:      LD      0,0(5) load id value
19:      LD      1,0(6) op: load left

```

这可以用两条指令代替：

```

LD 1,1(5) load id value
LD 0,0(5) load id value

```

它们具有同样的效果。

更多潜在的不足将由生成的测试和转移代码引起。不完整的笨例子是指令：

```

40:      LDA 7,0(7) jmp to end

```

这是一个煞费苦心的 `NOP`（一条“无操作”指令）。

然而，程序清单 8-14 的代码有一个重要理由：它是正确的。在匆忙提高生成代码的效率时，编译编写者忘记了这个原则并允许生成只有效率却不总是能正确执行的代码。这种行为如果不做好文档并预测可能会导致灾难。

由于学习所有改进编译器代码产生的方法超出了本书的范围，本章最后的两节将仅考察可以做出这些改进的主要范围和实现它们的技术，并简要说明某些方法如何用于 TINY 代码生成器来改进生成的代码。

程序清单 8-14 程序清单 8-1 示例程序的代码输出

```

* TINY Compilation to TM Code
* File: sample.tm
* Standard prelude:
0:      LD 6,0(0)      load maxaddress from location 0
1:      ST 0,0(0)      clear location 0
* End of standard prelude.
2:      IN 0,0,0      read integer value

```

```

3:      ST 0,0(5)      read: store value
* -> if
* -> Op
* -> Const
4:      LDC 0,0(0)      load const
* <- Const
5:      ST 0,0(6)      op: push left
* -> Id
6:      LD 0,0(5)      load id value
* <- Id
7:      LD 1,0(6)      op: load left
8:      SUB 0,1,0      op <
9:      JLT 0,2(7)      br if true
10:     LDC 0,0(0)      false case
11:     LDA 7,1(7)      unconditional jmp
12:     LDC 0,1(0)      true case
* <- Op
* if: jump to else belongs here
* -> assign
* -> Const
14:     LDC 0,1(0)      load const
* <- Const
15:     ST 0,1(5)      assign: store value
* <- assign
* -> repeat
* repeat: jump after body comes back here
* -> assign
* -> Op
* -> Id
16:     LD 0,1(5)      load id value
* <- Id
17:     ST 0,0(6)      op: push left
* -> Id
18:     LD 0,0(5)      load id value
* <- Id
19:     LD 1,0(6)      op: load left
20:     MUL 0,1,0      op *
* <- Op
21:     ST 0,1(5)      assign: store value
* <- assign
* -> assign
* -> Op
* -> Id
22:     LD 0,0(5)      load id value
* <- Id
23:     ST 0,0(6)      op: push left
* -> Const
24:     LDC 0,1(0)      load const
* <- Const
25:     LD 1,0(6)      op: load left
26:     SUB 0,1,0      op -

```

```

* <- Op
27:      ST 0,0(5)      assign: store value
* <- assign
* -> Op
* -> Id
28:      LD 0,0(5)      load id value
* <- Id
29:      ST 0,0(6)      op: push left
* -> Const
30:      LDC 0,0(0)      load const
* <- Const
31:      LD 1,0(6)      op: load left
32:      SUB 0,1,0      op = =
33:      JEQ 0,2(7)      br if true
34:      LDC 0,0(0)      false case
35:      LDA 7,1(7)      unconditional jmp
36:      LDC 0,1(0)      true case
* <- Op
37:      JEQ 0,-22(7)    repeat: jmp back to body
* <- repeat
* -> Id
38:      LD 0,1(5)      load id value
* <- Id
39:      OUT 0,0,0      write ac
* if: jump to end belongs here
13:      JEQ 0,27(7)    if: jmp to else
40:      LDA 7,0(7)      jmp to end
* <- if
* End of execution.
41:      HALT 0,0,0

```

8.9 代码优化技术考察

自从50年代出现第1个编译器以来，生成的代码质量一直受到重视。质量由目标代码的速度和大小来衡量，虽然通常速度更重要，现代编译器表明代码质量受编译过程中某些点的处理过程影响，这一系列步骤包括收集源代码信息然后利用这些信息执行代码改进变换（code improve transformation）改进代码中的数据结构，许多年来开发了大量的提高代码质量的技术，称为代码优化技术（code optimization techniques）。这个术语有些误导，由于仅在一些很特殊的情况下这些技术才能产生数学上的优化代码。不过这个名称很常用，我们还是继续使用它吧。

因为存在着许多代码优化技术，我们只能大概浏览一些最重要和最广泛使用的，甚至对这些也不给出细节和实现。本章有更多信息的参考书目。必须认识一点，编译器编写者不能希望包含每一种单个优化技术，而要根据语言的实际情况作出判断。哪种技术可以在增加最小编译器复杂度的情况下大大提高代码质量，有许多描述需要极复杂实现的优化技术的论文，而这些技术仅产生了相对有一点改进的目标代码（也就是减少一点运行时间）。经验通常显示一些基本方法虽然看起来是简单方式应用却可以带来重大提高，有时甚至减少一半或更多执行时间。

衡量某个优化技术的实现是否太复杂依赖实际代码改进的代价，不仅要确定实现的数据结构和额外代码开销的复杂度还要考虑优化步骤对编译器本身速度的影响。我们所学的所有分析

技术都与编译程序大小成正比。有些优化技术可能相对程序大小的平方或3次方增加编译时间,于是完全优化编译一个大程序时要延长好几分钟(最差时要几个小时)。这将导致用户回避使用优化(或不再使用这个编译器),用于实现优化的时间是巨大的浪费。

下面几节我们将先描述优化的主要来源然后介绍几种经典优化方法。接着是几种重要技术和实现的主要数据结构。自始至终都将给出简单示例来说明讨论的内容。下一节将给出更详细的例子说明讨论的一些技术如何应用于前面章节的TINY代码生成器,以及实现方法的建议。

8.9.1 代码优化的主要来源

下面将列出某些代码生成器不能产生好代码的地方,粗略地减少“代价”,也就是在这些地方代码可以获得多大改进。

1) **寄存器分配** 合理使用寄存器是高效代码的最重要特征。由于历史原因,可用寄存器很少——通常只8个或16个,这其中包括了特殊用途寄存器,如pc、sp和fp。这使得寄存器合理分配很困难,因为变量和临时变量竞争寄存器空间很激烈。这种情况仍存在于一些处理器中,特别是微处理器,解决这一问题的一个方法是可以增加直接在内存执行的操作的数量和速度,这样编译器一旦耗尽寄存器空间,就可以避免存储寄存器值到临时变量以释放寄存器值然后再装新值(称为寄存器溢出(register spill)操作)的代价。另一个方法(称为RISC方法)减少在内存直接执行的操作的数量(常常为0),但同时增加可用寄存器到32、64或128。在这种结构中,合理配寄存器变得至关重要,因为它可以保存全部或大部分全程变量在寄存器中。这种结构中分配寄存器失误的代价是频繁装载和存入值。同时因为有了很多可用的寄存器,寄存器分配工作也变得简单了。这样,提高代码质量的重要努力应着眼于合理分配寄存器。

2) **不必要操作** 代码改进的第2个主要来源是避免产生冗余或不必要的操作代码。这种优化从很简单的搜索局部代码到分析整个程序的语法特性各不相同。确认这些操作的方法很多,也有许多对应技术。这种优化的一种典型例子是代码中重复出现的表达式,而且它们的值相同,可以保存的第1次的计算值并删除重复计算(称为公共子表达式消除(common subexpression elimination))^①。另一个例子是避免存储不再使用的变量或临时变量的值(这要与前面的优化共同进行)。

全程的优化涉及识别不可到达(unreachable)或死代码(dead code),典型例子是使用常量标志开关调试信息:

```
#define DEBUG 0
...
if (DEBUG)
{...}
```

如果DEBUG设为0(如代码中所示),那么在if语句的大括号内的代码将不可到达,于是不必产生这段目标代码,甚至连if语句也可以省掉。另一个不可到达代码例子是不调用的过程(或只在不可到达的代码处调用)的消除,不可到达代码不总是对速度产生重大影响,不过可以真正减少目标代码大小,这是值得的,特别是当分析中很小的代价就可以识别大部分明显的情况。有时为了识别不必要操作,用代码生成器处理然后检测目标代码的冗余更容易些。一种情况是对在生成跳向表示结构控制语句时产生冗余作出预测是很困难的。这些代码包含转移到相邻的下一

① 一个好的程序员可以避免公共表达式在源码中这样扩散,读者不应认为优化只是用于帮助差劲的编程者。许多来自地址计算的公共子表达式由编译器产生,并不能由好的源代码来消除它们。

条语句或转移到转移语句。转移优化(jump optimization)步骤可以去除这些不必要的转移。

3) 高代价操作 代码生成器不只要寻找不必要操作,还要利用机会减少必要操作的代价。要用比源代码更简单实现更低的代价实现操作。典型例子用低代价操作代替算术操作,例如,乘2可以用移位操作实现。小整数数据幂,比如 x^3 ,可以用连乘 $x*x*x$ 实现。这种优化称为减轻强度(reduction in strength),它可以扩展到许多方面,比如将涉及小的整数乘法替换为移位和加法(例如用 $2*2*x$ 代替 $5*x + x$ ——两个移位和一个加法)。

一个相关的优化是用有关常数信息删除可能的操作或预先计算一些操作。例如,两个常数相加,比如 $2+3$,可以由编译器计算并作常数5代替(这称为常数数据合并(constant folding))。

有时有必要确定一个变量在程序局部或全程是否有恒定值,这样变换就可应用于涉及该变量的表达式(称为常量传播(constant propagation))。

有时相对昂贵的操作是过程调用,这里许多调用操作序列必须执行。现代处理器通过提供支持标准调用的硬件减少了这些代价。但是去除频繁调用小过程还是能产生可观的加速。有两种标准方法可以去掉过程调用。一个是用过程体代替过程调用(使用合适的参数代替形式参数)。这称为过程内嵌(procedure in lining),有时这是语言选项比如C++。另一个消除调用的方法是识别尾部递归(tail recursion),也就是过程最后的操作是调用自身,例如,过程:

```
int gcd( int u, int v)
{ if (v==0) return u;
  else return gcd(v,u % v); }
```

是尾递归的,而过程

```
int fact( int n )
{ if (n==0) return 1;
  else return n * fact(n-1); }
```

则不是。尾递归等同过将新的调用参数赋给形式参数并转移到过程体的开始。例如尾递归过程gcd可以被编译器重写为等效代码:

```
int gcd( int u, int v)
{ begin:
  if (v==0) return u;
  else
  { int t1 = v, t2 = u%v;
    u = t1; v = t2;
    goto begin;
  }
}
```

(注意代码中临时变量的微妙利用)。这个处理称为尾递归消除(tail recursion removal)。

要提及的问题是过程调用与寄存器分配有关。过程调用之前必须准备保存和恢复在调用内部使用的寄存器。如果提供的是多寄存器分配,将增加过程调用代价,因为更多的寄存器要保存和恢复。有时在寄存器分配中包含调用考虑会减少花费^①。但这是一个普遍现象:有时优化会引起反面效果,因此必须考虑取舍。

在代码生成的最后阶段,通过使用目标机器上的特殊指令可以减少某些操作的代价。例如,许多结构包括块移动操作比单独拷贝或数组元素要快许多。还有地址计算有时可以优化,当结

^① Sun SparcStation中的寄存器窗口表示硬件支持过程调用的寄存器分配的一个例子。参见8.6.2节最后。

构允许几种地址模式或偏移计算结合到一条指令中时。同样，用于索引的自动增量和减量也是很实用的（VAX 结构甚至有为循环设的增量比较分支）。这些优化来自先前的指令选择(instruction selection)或机器语言使用(use of machine idioms)。

4) 预测程序行为 为了实施前面描述的一些优化，编译器必须收集有关程序中变量、值和过程使用的信息：表达式是否重用(可变成公共子表达式)、变量是否改变、何时改变或一直不变、过程是否调用。编译器必须在计算技术范围内作最坏的假设，收集的信息有可能产生错误的代码：变量在特定点可能恒定也可能不恒定、编译器必须假设它不恒定，这意味着必须将编译器做成不是最优化的，甚至常常对程序的行为信息利用较差。实际上对程序的分析越透彻，代码优化器可以得到的信息越多。然而，即使今天最先进的编译器也会无法发现程序中的一些可改进之处。

另一个许多编译器采用的方法是：实际运行中采集程序行为的统计信息，然后用于预测哪条分支最有可能运行、哪个过程经常调用以及哪部分代码最经常执行。这些信息可以用于调整转移结构，循环和过程代码来最小化最经常执行部分的运行时间，当然这个处理要求类似梗概编译器(profiling compiler)访问适当的数据以及(至少一部分)包含产生这些数据的指示代码的可执行代码。

8.9.2 优化分类

由于有许多优化方法和技术，有必要采用不同分类规划强调优化的不同质量以减少学习难度。两个有用的分类是在编译过程中何时可以应用优化和优化应用于程序的哪些部分。

首先我们考虑应用程序在编译过程中的时间。优化可以在编译的每阶段分别执行。例如，常数合并可以在分析时进行(虽然通常是迟一些，这样编译器可以得到与源代码相同的表示)。另一方面，一些优化可以延迟到目标代码生成之后—检查并重写目标代码以反映优化。例如，转移优化可以以这种方式进行(有时因为通常只观察目标代码的一小部分来进行优化，所以在目标代码上进行的优化称之为窥孔优化(peephole optimization))。

通常，主要的优化工作在中间代码生成部分、中间代码生成后或目标代码生成部分。为了使优化不依赖于目标机器的特性(称为源代码级优化(source-level optimization))，可以在依赖目标机器结构的动作(目标代码级优化(target-level optimization))之前执行。有时一种优化可以同时有源码级部分和目标码级部分。例如，在寄存器分配中，经常要计算变量引用次数并将高引用率的变量放入寄存器。这个任务又分成了一个源码级部分，这里为选择的变量分配寄存器不必知道有多少可用寄存器。然后寄存器赋值步骤依赖于目标机器为这些标记的变量分配实际的寄存器，或到称为伪寄存器(pseudoregister)的内存(万一没有可用寄存器的话)。

在不同的优化中，考虑某一优化对其他优化产生的影响相当重要。例如，应在执行不可到达代码消除之前进行传播常量操作，这是因为在测试变量被发现是常数据之后，某些代码会变得不可大到达。在偶然情况下会发生两种优化无法为对方发现进一步优化机会的阶段问题(phase problem)。如下例

```
x = 1;
...
y = 0;
...
if (y) x = 0;
...
if (x) y = 1;
```

首次常量传播会产生如下代码

```
x = 1;  
...  
y = 0;  
...  
if (0) x = 0;  
...  
if (x) y = 1;
```

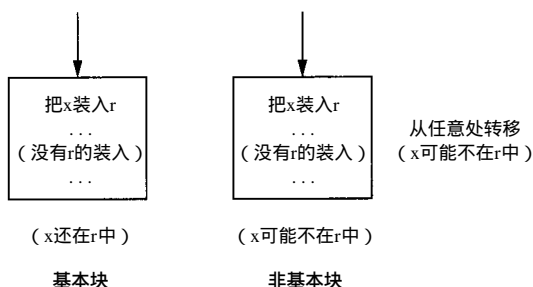
现在第1个if体变成不可到达；消除之

```
x = 1;  
...  
y = 0;  
...  
if (x) y = 1;
```

现在可以进行进一步的常量传播和不可到达代码消除步骤。由于这个原因，一些编译器反复执行几组优化以确保大部分可以利用的优化机会已经找到。

第2类优化考虑的是优化应用的程序范围。这类优化还分为局部 (local)、全程(global)和过程间(interprocedural)优化。局部优化定义为应用于代码的线性部分 (straight-line segment of code)的优化，也就是代码中没有跳进或跳出语句^①。一个最大的线性代码序列称为基本块 (basic block)。局部优化的定义限制这些基本块。扩展超出基本块，但限制在单个过程中的优化称为全程优化(因为限制在过程中，所以这不是真正的“全程”)。优化扩展出过程边界到整个程序称为过程间优化。

局部优化相对易于进行，因为代码的线性特征允许信息以简单方式下传。例如，基本块中前一条指令将一个变量值装入寄存器，只要寄存器不被再次装入，就可以在块的后面继续认为值还存在。这个结论在转移的干扰代码时就不正确了。如下图所示：



全程优化相对要困难一些，通常要求一种称为数据流分析(data flow analysis)的技术，它试图透过转移边界收集信息。过程间优化更困难，因为要涉及可能不同的参数传递机制，非局部变量访问以及计算相同调用的过程的同步信息。过程间优化的另一个复杂之处是许多过程可能分别编译最后才链接到一起。于是编译器在没有链接程序基于编译阶段收集的信息的基础上得出的优化信息时不能进行优化。由于这个原因，许多编译器只执行很基本的过程间分析或者根本就不执行。

全程优化的一个特别部分是循环。由于循环通常多次执行，应该特别注意循环内部的代码，

^① 过程调用表示一种特殊跳转，通常它们打断直线执行代码。然而，由于它们总是返回到相邻的下一条指令，经常可以包含在直线代码中间并由代码生成过程以后处理。

特别是减少复杂运算。典型的循环优化策略着眼于识别每次执行增长值固定的变量（也称为归纳变量(induction variable)）。这包括循环控制变量和其他依赖于循环变量的变量，选择的归纳变量可以放入寄存器，使其计算简化。这些代码重写包括从循环中删除常量计算（称为代码移动(code motion)）。实际上，重新安排代码也有利于提高基本块中的代码效率。通常，循环优化的额外任务是区分程序中的循环，然后就可以进行优化了。因为缺乏结构化控制和使用 goto 实现循环，这种循环发现(loop discovery)是必要的。虽然循环发现在少数语言(如FORTRAN)中需要，但在大部分语言中，语法本身可以用来定位循环结构。

8.9.3 优化的数据结构和实现技术

一些优化可以在语法树的变换上实现。这些包括常数合并和不可到达代码消除，通过删除或以简单形式替换相应的子树来实现。用于后来优化的信息也可以在构造和遍历语法树时收集，比如引用次数和其他有用信息，保存到树的属性或符号表项中。

对于前面提到的一些优化，语法树不广泛或结构不适合于收集信息并进行优化。作为代替执行全程优化的优化器使用从中间代码构造的过程图形表示，称为流图(flow graph)。流图的节点为基本块，边则是来自条件或非条件转移(目的是作为其他基本块的开始)。每个基本块节点包含块的中间代码序列。例如，图 8-4 给出了对应程序清单 8-2 中间代码的流图(图中基本块标记为了以后引用)。

流图以及每个基本块都可以从中间代码一次遍历中构造完成。每个新的基本块识别如下^①：

- 1) 第1条指令开始一个新基本块。
- 2) 每个转移目的标签开始一个新基本块。
- 3) 每条跟随在转移之后的指令开始一个新基本块。

可以为还没达到的向前转移构造新的空节点，并插入到符号表的标号名下，以备标号到达时查找。

流图是数据流分析的主要数据结构，积累信息用于优化。不同的信息要求对流图作不同处理，而且收集的信息各不相同，对应于不同的优化要求。因为没有足够的空间在这个概览中描述数据流分析技术的细节(参见本章尾部“注意与参考”小节)，描述这种过程可以处理的数据的例子将是很有意义的。

计算所有变量的可达定义(reaching definition)集是一个标准的数据流分析问题，变量在每个基本块的开始处。这里一个定义是一条中间代码指令，可以设置变量值，比如赋值或读入^②。例

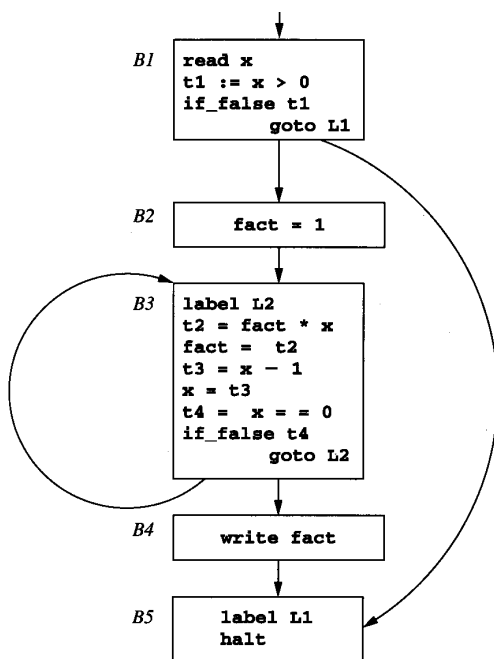


图8-4 程序清单 8-2 中间代码的流图

① 这个标准允许过程调用包含在基本块中。由于调用不对流图增加新路径，所以这是可行的。然后，当基本块分别处理时，调用可以在需要时分离出来特别处理。

② 请不要与C定义混淆，这是一种声明。

如,图8-4中定义的变量 `fact` 是基本块 $B2(fact=1)$ 中的一条指令以及块 $B3(fact=t2)$ 中的第3条指令。让我们调用定义 $d1$ 和 $d2$ 。如果在块开始处变量保持定义时建立的值,则这个定义称为到达(reach)基本块。图8-4的流图中,可以建立 `fact` 的定义到达 $B1$ 或 $B2$, $d1$ 和 $d2$ 到达 $B3$ 以及只有 $d2$ 到达 $B4$ 和 $B5$ 。到达定义可以用于许多优化—常数传播。例如,如果到达块唯一定义为单个常数值,那么这个变量可以用这个值来代替(至少在块中另一个定义到达前)。

流图很适用于表示有关每个过程的全程信息,不过基本块仍旧用简单代码序列表示。一旦执行数据流分析,每个基本块的代码也产生了,另一个数据结构经常被构造出来,称为基本块的DAG (DAG of a basic block)(DAG = directed acyclic graph直接非循环图)(DAG可以在没有构造流图时为每个基本块构造)。

DAG数据结构跟踪基本块中值和变量的计算和赋给。块中使用的来自别处的值表示为叶子节点。其上的操作和其他值表示为内部节点。赋给新值通过把目标变量或临时变量的名字附加到表示赋值的节点上来表示(416页描述了这种结构的一个特例)[⊖]。

例如,图8-4中基本块 $B3$ 可以用图8-5中的DAG表示(基本块开头和尾部的转移的标号通常包含在DAG中)。注意在这个DAG中拷贝操作如 `fact=t2` 和 `x=t3` 不创建新节点,而只是简单地用标号 $t2$ 和 $t3$ 为节点加上新标签。还要注意标为 x 的叶子节点有两上父节点,其原因在于外来值 x 用于两条分别的指令中。这样重复使用同一个值也在DAG结构中表示出来了。DAG的这个特点允许它表示公共子表达式的重复使用。例如C赋值语句:

```
x = (x+1)*(x+1)
```

翻译成三地址指令

```
t1 = x + 1
t2 = x + 1
t3 = t1 * t2
x = t3
```

图8-6给出这个指令序列的DAG,显示了表达式 $x+1$ 的重复使用。

基本块的DAG可以通过维护两个目录来构造。第1个是包含变量名和常数的表,带有可返回当前赋值变量的DAG节点的查找操作(符号表可以用作这个表)。第2个是DAG节点表,带有给出操作和子节点的查找功能,返回操作和孩子节点,若没有则返回空。这个操作允许查找已存在的值,不用在DAG中构造新节点。例如,一旦图8-6中的+节点及其子节点 x 和 1 被构造并分配名字 $t1$ (作为三地址指令 $t1=x+1$ 处理的结果)。在第2个表中查找(+、 x 、 1)将返回这个已构造的节点,三地址指令 $t2=x+1$ 只是使 $t2$

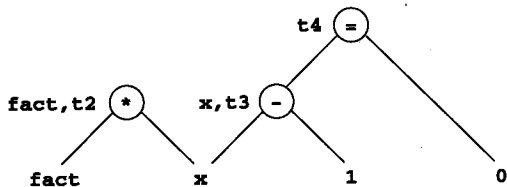


图8-5 图8-18中基本块 $B3$ 的DAG

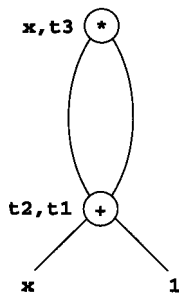


图8-6 与C赋给 $x = (x+1)*(x+1)$ 相对应的三地址指令的DAG

⊖ 这个DAG结构的描述适合使用三地址指令作为中间代码,不过类似的DAG可以定义用于P-代码或目标汇编代码。

也被赋给这个节点。这个构造的细节可以在别处找到(参见“注意与参考”部分)。

目标代码或修订过的中间代码,可以通以一种可能的拓朴顺序遍历 DAG的非叶子节点来产生(DAG的拓朴顺序是一种遍历,节点的孩子先于双亲被访问)。因为有多种拓朴顺序,可以从 DAG中产生许多不同代码序列。哪一种更好依赖于多种因素,包括目标机器结构的细节。例如,图8-5中3个非叶子节点的一个正规遍历序列将产生下面的三地址指令。这将代替原始的基本块:

```
t3 = x - 1
t2 = fact * x
x = t3
t4 = x == 0
fact = t2
```

当然,我们会希望避免使用临时变量,于是要产生下面的等价三地址码,其顺序必须固定:

```
fact = fact * x
x = x - 1
t4 = x == 0
```

图8-6中的DAG的一个类似遍历产生下示修正三地址码

```
t1 = x + 1
x = t1 * t1
```

使用DAG的基本块产生目标代码,自动得到局部公共子表达式的消除。DAG表示法也使消除冗余存储(赋值)成为可能并告诉我们对每个变量的引用次数(节点的父节点数表示引用数)。这为合理分配寄存器提供了有用的信息(例如,假设一个值有多个引用,则放入寄存器;如果看到了所有引用,这个值已消亡不必再维护了等等)。

最后一个常用于辅助寄存器分配作为代码生成中数据维护的方法称为寄存器描述器(register descriptor)和地址描述器(address descriptor)。寄存器描述器为每个寄存器建立一个列表,表中列出当前值在寄存器中的变量名(当然,在那一点它们有同样的值)。地址描述器则为每个变量与内存地址建立联系。这些可以是寄存器(这种情况下,变量可以在相应的寄存器描述器中找到)或内存或两者都有(如果变量刚从内存中装入寄存器,但值尚未改变)。这种描述器允许跟踪值在内存和寄存器之间的移动,并可以重用已装入寄存器的值,还有回收寄存器,不管是不再包含后续使用变量的值或将值存入到适当内存位置中(溢出操作)。

例如,图8-5中的基本块 DAG,并考虑,使用3个寄存器0、1和2对应从左到右遍历内部节点产生TM代码。假设有4个地址描述:inReg(reg-no)、isGlobal(global-offset)、isTemp(temp-offset)和isConst(value)(这对应于TM机上的TINY运行时刻环境,见前面章节的讨论)。进一步假设x在全程位置0, fact在全程位置1,全程位置通过gp寄存器访问,临时变量位置则通过mp寄存器访问。最后,假设所有寄存器中都已无初始值,这样在基本块代码产生开始之前,变量和常量的地址描述器如下所示:

变量/常量	地址描述器
fact	isGlobal(1)
x	isGlobal(0)
t2	-
t3	-
t4	-
1	isConst(1)
0	isConst(0)

寄存器描述器表为空，就不再列出了。

现在假设产生了如下代码：

```
LD 0,1(gp) load fact into reg 0
LD 1,0(gp) load x into reg 1
MUL 0,0,1
```

那么地址描述器就变成

变量/常量	地址描述器
fact	inReg(0)
x	isGlobal(0),inReg(1)
t2	intReg(0)
t3	-
t4	-
1	isConst(1)
0	isConst(0)

寄存器描述器则为

寄存器	包含的变量
0	fact, t2
1	x
2	-

现在给出后续代码

```
LDC 2,1(0) load constant 1 into reg 2
ADD 1,1,2
```

地址描述器变成：

变量/常量	地址描述器
fact	inReg(0)
x	inReg(1)
t2	inReg(0)
t3	inReg(1)
t4	-
1	isConst(1),intReg(2)
0	isConst(0)

寄存器描述器变成：

寄存器	变量/常量
0	fact, t2
1	x, t3
2	1

我们把计算DAG中剩余节点值的代码以及描述结果地址和寄存器描述器留给读者完成。

我们的代码优化技术概览到为止。

8.10 TINY代码生成器的简单优化

8.8节中给出的TINY语言代码生成器产生的代码效率很低。这可以从程序清单 8-14的42条指令与程序清单8-13中9条手写的等价程序指令的比较中看出来。基本上，低效率有两个来源：

- 1) TINY代码生成器对TM机的寄存器的运用较差(实际上，从来不使用寄存器2、3或4)。
- 2) TINY代码生成器为测试产生不必要的逻辑值 0和1，由于这些测试只出现在 if语句和 while语句中，简单代码即可实现。

本节希望指出相对粗糙的技术如何实质地提高 TINY编译器产生的代码的性能。实际上，不必生成基本块或流图而是直接从语法树产生代码。唯一需要的机制是附加的属性数据和一些稍显复杂的编译代码。我们不给出此处描述的改进的实现细节，还是留给读者练习。

8.10.1 将临时变量放入寄存器

我们首先描述的优化是一个简单方法，把临时变量保存在寄存器中，而不是经常操作内存读出和写入。在TINY代码生成器中临量变量总是存在位置：

```
tmpOffset(mp)
```

这里tmpOffset是初始的静态值，每次临时变量存储后递减，而每次读出后则递增(见附录B，第2033行和第2027行)。将寄存器作为临时变量存储位置的一个简单方法是把 tmpOffset翻译成初始指向寄存器，只在可用寄存器用完时使用实际的内存偏移。例如，假设我们要将所有可用寄存器用于临时变量(除pc、gp和mp)，那么 tmpOffset值0~4将翻译成寄存器0~4的引用，当值从-5开始时就使用偏移(在值上加上5)。这种机制可以直接应用于代码生成器的相应测试，可封装进辅助过程(将命名为 saveTmp和loadTmp)。还要注意，在产生递归代码后子表达式的计算结果应保存除0外的其他寄存器中。

有了这些改进，TINY代码生成器产生如程序清单8-15所示的TM代码序列(请与程序清单8-14比较)。这个代码缩短20%并没有临时变量的存储(也没有使用寄存器5，即mp)。寄存器2、3和4仍没有使用。这并不奇怪：程序中的表达式很少复杂到同时需要两个或3个临时变量。

程序清单8-15 临时变量保存在寄存器中的示例 TINY程序的TM代码

0:	LD	6,0(0)	17:	ST	0,1(5)
1:	ST	0,0(0)	18:	LD	0,0(5)
2:	IN	0,0,0	19:	LDC	1,1(0)
3:	ST	0,0(5)	20:	SUB	0,0,1
4:	LDC	0,0(0)	21:	ST	0,0(5)
5:	LD	1,0(5)	22:	LD	0,0(5)
6:	SUB	0,0,1	23:	LDC	1,0(0)
7:	JLT	0,2(7)	24:	SUB	0,0,1
8:	LDC	0,0(0)	25:	JEQ	0,2(7)
9:	LDA	7,1(7)	26:	LDC	0,0(0)
10:	LDC	0,1(0)	27:	LDA	7,1(7)
11:	JEQ	0,21(7)	28:	LDC	0,1(0)
12:	LDC	0,1(0)	29:	JEQ	0,-16(7)
13:	ST	0,1(5)	30:	LD	0,1(5)
14:	LD	0,1(5)	31:	OUT	0,0,0
15:	LD	1,0(5)	32:	LDA	7,0(7)
16:	MUL	0,0,1	33:	HALT	0,0,0

8.10.2 在寄存器中保存变量

进一步的改进可以将 TM 的一些寄存器用于变量存储。这比前面的优化所做的工作要多，因为变量位置必须在代码生成和存储符号表之前确定。一个基本方案是简单选取几个寄存器存放程序中最常用的几个变量。为了确定哪些变量“最常用”，必须给出引用次数(使用和赋值)。在循环中引用的变量(在循环体或测试表达式中)应优先考虑，因为引用在循环执行时将重复进行。在许多现在编译器中工作出色的一个简单方法是将所有循环内引用都乘以 10，在两层嵌套循环中乘 100，如此类推。引用计数可以在语法分析时完成。之后作出分别的变量传递，符号表中储存的位置属性必须能表明那些定位于寄存器的变量与定位于内存的变量的差别。一个简单的方案使用枚举类型指示变量位置：本例中，只有两种可能 `inReg` 和 `inMem`。另外，第 1 种情况要记录寄存器号，第 2 种情况要记录内存地址(这是变量地址描述器的一个简单例子：不需要寄存描述器，因为它们代码生成过程中保持不变)。

有了这些改变，示例程序的代码将使用寄存器保存变量 `x`，寄存器 4 保存 `fact` (这里只有两个变量，所以可以都放入寄存器)。假设寄存器 0 到 2 仍保留给临时变量使用。对程序清单 8-2 代码的修改给出在程序清单 8-16 中。这段代码又比前面代码缩短许多，不过仍比手写的代码长。

程序清单 8-16 临时变量和变量保存在寄存器中的示例 TINY 程序的 TM 代码

0:	LD	6,0(0)	13:	LDC	0,1(0)
1:	ST	0,0(0)	14:	SUB	0,3,0
2:	IN	3,0,0	15:	LDA	3,0(0)
3:	LDC	0,0(0)	16:	LDC	0,0(0)
4:	SUB	0,0,3	17:	SUB	0,3,0
5:	JLT	0,2(7)	18:	JEQ	0,2(7)
6:	LDC	0,0(0)	19:	LDC	0,0(0)
7:	LDA	7,1(7)	20:	LDA	7,1(7)
8:	LDC	0,1(0)	21:	LDC	0,1(0)
9:	JEQ	0,15(7)	22:	JEQ	0,-12(7)
10:	LDC	4,1(0)	23:	OUT	4,0,0
11:	MUL	0,4,3	24:	LDA	7,0(7)
12:	LDA	4,0(0)	25:	HALT	0,0,0

8.10.3 优化测试表达式

我们讨论的最后一个优化是简化生成的 `if` 语句和 `while` 语句代码。因为这些表达式产生的代码很通用，布尔值真和假应用为 0 和 1，尽管 TINY 没有布尔变量且不需要这种通用级别。这还导致了额外的装入常量 0 和 1，以及由 `genStmt` 代码独立产生用于控制语句的额外测试。

此处描述的改进依赖于比较操作符必须为测试表达式的根节点。这个操作符的 `genExp` 代码只是简单地产生代码将左操作数减去右操作数，把结果放入寄存器 0。 `if` 语句或 `while` 语句的代码将检查使用了哪个比较算符并产生相应的条件转移代码。

这样，程序清单 8-16 中 TINY 代码

```
if 0<x then.
```

现在对应于 TM 代码

```

4: SUB 0,0,3
5: JLT 0,2(7)
6: LDC 0,0(0)
7: LDA 7,1(7)
8: LDC 0,1(0)
9: JEQ 0,15(7)

```

将由简单的TM代码代替

```

4: SUB 0,0,3
5: JGE 0,10(7)

```

(注意：假情况转移必须补充条件JGE到测试算符<中)。

有了这个优化，为测试程序生成的代码变成了程序清单 8-17所示(我们还在这一步骤中包括了去除代码尾部的空转移，对应于TINY代码中无else部分的if语句。这只要在genStmt中增加对if语句的简单检查即可)。

程序清单8-17的代码相对接近手写代码了。即使如此，还有一些特殊情况可以优化，这将在练习中。

程序清单8-17 变量与临时变量放入寄存器并简化表达式测试的示例 TINY程序的TM代码

0: LD 6,0(0)	9: LDC 0,1(0)
1: ST 0,0(0)	10: SUB 0,3,0
2: IN 3,0,0	11: LDA 3,0(0)
3: LDC 0,0(0)	12: LDC 0,0(0)
4: SUB 0,0,3	13: SUB 0,3,0
5: JGE 0,10(7)	14: JNE 0,-8(7)
6: LDC 4,1(0)	15: OUT 4,0,0
7: MUL 0,4,3	16: HALT 0,0,0
8: LDA 4,0(0)	

练习

8.1 为Lex表示法中的C注释写出一个正则表达式(提示：参见2.2.3节中的讨论)。给出对应下面算术表达式的三地址指令序列：

- $2+3+4+5$
- $2+(3+(4+5))$
- $a*b+a*b*c$

8.2 给出对应前一个练习中算术表达式的P-代码序列。

8.3 给出对应下列C表达式的P-代码指令：

- $(x = y = 2) * (x = 4)$
- $a(a)i = b(i = n)$
- $p \rightarrow next \rightarrow next = p \rightarrow next$

(假设相应的结构定义。)

8.4 给出前一练习中表达式的三地址指令。

8.5 给出对应下列TINY程序的(a)三地址码或(b)P-代码

```

{ Gcd program in TINY language }
read u;

```

```

read v; { input two integer }
if v = 0 then v := 0 { do nothing }
else
  repeat
    temp := v;
    v := u - u/v*v; { computes u mod v }
    u := temp
  until v = 0
end;
write u { output gcd of original v & v }

```

- 8.6 参照程序清单8-4的四元式给出程序清单8-5三元式的C数据结构定义。
- 8.7 扩展表8-1P-代码的属性文法(8.2.1节)成 a 8.3.2节的子描述语法; b 8.4.4节的控制结构语法。
- 8.8 对表8-2的三地址码属性文法重复上面练习。
- 8.9 描述代码生成如何采用6.5.2节的普通遍历过程,这是否有意义?
- 8.10 增加地址操作符&和* (用C语法)以及二元结构域选择操作符.到
- a. 8.2.1节的表达式语法。
 - b. 8.2.2节的语法树结构。
- 8.11 a. 为8.4.4节的控制语法添加repeat-until(或do-while语句),并画出对应图8-2的合适控制图。
- b. 为语法(8.4.4节)重写语法树结构定义以包含a部分的新结构。
- 8.12 a. 描述如何系统地将for语句转换成对应的while语句,用于产生代码是否可行?
- b. 描述如何系统地将case或switch语句转换嵌套的if语句,用于产生代码是否可行?
- 8.13 a. 参照图8-2的Borland 80 x 86 C编译器显示的循环结构画出控制图。
- b. 参照图8-2为8.6.2节中Sun SparcStatin C编译器显示的循环结构画出控制图。
 - c. 假设一个条件转移执行时间3倍于代码“穿过”(比如条件为假)。那么a和b部分的转移组织是否比图8-2有时间优势?
- 8.14 一种代替case或switch语句为每个case顺序测试的实现称为转移表(jump table),其中case索引被用于索引转移的偏移对应到绝对转移。
- a. 这种实现方法只有在大量不同case在相对较小的索引范围内密集发生时才有优势,为什么?
 - b. 代码生成器只是在超过10个case时才产生这种代码。确定你的C编译器是否有一个最小值决定是否产生switch语句的转移表。
- 8.15 a. 开发类似于8.3.2节的多维数组元素地址计算公式。说明你的所有假设。
- b. 假设有如下用C代码定义的数组变量a

```
int a[12][100][5]
```

假设一个整数在内存中占两个字节。用你在a部分中的公式确定下面变量相对a基址的偏移

```
a[5][42][2]
```

- 8.16 参照8.5.2节的函数定义/调用语法给出下面程序:

```

fn f(x)=x+1
fn g(x,y)=x+y

```

g f(x(3), 4+5)

- a. 写出程序清单的genCode过程为该程序产生的P-代码指令序列。
 - b. 写出此程序的三地址指令代码。
- 8.17 文中没有指出arg三地址指令在函数调用中是否使用：有些版本的三地址码要求所有arg语句混合出现(参见8.5.1节)。讨论这两种方法的利弊。
- 8.18 a. 列出本章所用的全部P-代码指令，以及其意义和使用的描述。
b. 列出本章所用全部三地址指令，以及意义和使用的描述。
- 8.19 写出练习8.5中TINY gcd程序的等价TM程序：
- 8.20 a. TM没有寄存器到寄存器移动指令，说明这是如何实现的。
b. TM没有调用和返回指令，说明如何模拟实现。
- 8.21 为TM设计一个浮点协处理器，可以在不改变现存寄存器和内存定义的情况下使用(参见附录C)。
- 8.22 写出TINY编译器为下列TINY表达式和赋值产生的TM指令序列：
- a. $2+3+4+5$
 - b. $2+(3+(4+5))$
 - c. $x := x + (y + 2 * z)$, 假设x, y和z分别在dMem位置0、1和2。
 - d. $v := u - u / v * v$;
- (来自练习8.5 TINY gcd程序中的一行；假设标准的TINY运行时环境)。
- 8.23 为8.5.2节的函数调用设计TM运行时环境。
- 8.24 Borland 3.0编译器产生如下80×86代码来计算 $x < y$ 比较的逻辑结果，假设x和y为整数，在本地活动记录中的偏移为-2和-4：

```

mov     ax, word ptr [bp-2]
cmp     ax, word ptr [bp-4]
jge     short @1@86
mov     ax, 1
jmp     short @1@114
@1@86:
xor     ax, ax
@1@114:

```

请与TINY编译器为同一表达式产生的TM代码比较。

- 8.25 检查你的C编译器如实现短回路布尔操作，并与8.4节中的控制结构实现比较。
- 8.26 为练习8.5中TINY gcd程序对应的三地址码画出流图。
- 8.27 为练习8.5中TINY gcd程序的repeat语句体的基本块画出DAG。
- 8.28 考虑8.9.3节的图8-5的DAG，假设最右节点的相等操作符在TM机中用相减模拟，对应这个节点的TM代码将如下所示：

```

LDC 2,0(0) load constant 0 into reg 2
SUB 2,1,2

```

写出执行上述指令后寄存器和地址描述器(基于8.9.3节)。

- 8.29 确定你的C编译器执行的优化，并与8.9节描述比较。
- 8.30 两个附加的优化可以应于TINY代码生成器，如下所示：
 - 1) 如果测试表达式的一个操作数为常数0，则在产生跳条件转移之前不必执行相减。
 - 2) 如果一条赋值语句的目标已在寄存器中，则右边表达式可以计算到此寄存器，这

样节省了寄存器到寄存器移动。

给出这两个优化增加到代码生成器中后,由示例 TINY 程序产生的代码,这些代码与手写的程序清单 8-13 中的代码相比如何?

编程练习

- 8.31 重写程序清单 8-7(8.2.2 节)中的代码来产生 P-代码作为同步字符串属性对应于表 8-1 的属性文法,并与程序清单 8-7 中代码比较复杂度。
- 8.32 重写下列 P-代码产生过程来生成三地址指令:
- 程序清单 8-7(简单 C 表达式)。
 - 程序清单 8-9(带数给的表达式)。
 - 程序清单 8-10(控制语句)。
 - 程序清单 8-11(函数)。
- 8.33 写出类似程序清单 8-8 的 Yacc 说明,对应以下代码生成过程。
- 程序清单 8-9(带数组的表达式)。
 - 程序清单 8-10(控制语句)。
 - 程序清单 8-11(函数)。
- 8.34 重写程序清单 8-8 的 Yacc 说明出产生三地址码代替 P-代码。
- 8.35 为程序清单 8-7 的代码生成过程增加练习 8.10 中的操作符。
- 8.36 重写程序清单 8-10 的代码生成过程以包含练习 8.11 中的新控制结构。
- 8.37 重写程序清单 8-7 的代码,产生 TM 代码代替 P-代码(假设代码生成实用程序在 TINY 编译器的 code.h 文件中)。
- 8.38 使用练习 8.23 中设计的运行时环境重写程序清单 8-11 的代码以产生 TM 代码。
- 8.39 a. 为 TINY 语言和编译器增加简单数组。这要求在语句之前添加数组定义,如

```
array a[10];
i := 1;
repeat
    read a[i];
    i := i + 1;
until 10 < i
```

b. 为(a)部分代码添加边界检查,于是越界下标将引起 TM 机停机。

- 8.40 a. 实现练习 8.21 中设计的 TM 浮点协处理器。
- b. 用 TM 的浮点能力将 TINY 语言和编译器中的整数替换为实数。
- c. 重写 TINY 语言和编译器使之同时包含整形和浮点值。
- 8.41 编写一个 P-代码到三地址码的转换器。
- 8.42 编写一个三地址码到 P-代码的转换器。
- 8.43 重写 TINY 代码生成器以生成 P-代码。
- 8.44 编写 P-代码到 TM 机代码的转换器,假设有前面练习中描述的 P-代码生成器以及文本描述的 TINY 运行时环境。
- 8.45 重写 TINY 代码生成器以产生三地址码。
- 8.46 编写三地址码到 TM 代码的转换器,假设有前面练习中的三地址码生成和文本描述的 TINY 运行时环境。

8.47 实现8.10节中描述的3种TINY代码生成器优化：

- a. 将前3个TM寄存器用于临时变量。
- b. 将寄存器3和4用于最常用变量。
- c. 优化测试表达式代码，不再产生布尔值0和1。

8.48 在TINY编译器中实现常数合并。

8.49 a. 实现练习8.30中优化1。

- b. 实现练习8.30中优化2。

注意与参考

代码生成和优化技术有很多；本章只是一个介绍。这些技术（特别是数据流分析）的概览，理论观点上的把握，包含在AHO、Sethi和Ullman(1986)。一些更详细的部分主题参见Fischer和LeBlanc(1991)。case/switch语句的转移表(8.14)也有描述。专门处理器(MIPS、Sparc和PC)代码生成的例子见Fraser和Hanson(1995)。代码生成作为属性分析在Slonneger和Kurtz(1995)。

自从第1个编译器出现，中间代码随编译器不同而不同成为移植问题的一个来源。最初，认为可以开发出通用中间代码用于所有编译器并解决移植问题 (Strong(1958)、Steel(1961))。不幸的是没有取得进展，三元式和四元式是中间代码的传统形式并用于许多编译器。P-代码详细描述在Nori et al(1981)。一种更先进的P-代码称为U-代码。允许更好的目标代码优化，在Perkins和Sites(1977)中描述。一个类似版本的P-代码在Modula-2优化编译器中使用(Powell(1984))。一个特殊的用于Ada编译器的中间代码，称为Diana，在Goos和Wulf(1981)中有描述。一种使用LISP风格前缀表达式的称为寄存器转换语言或RTL的中间代码在GNU编译器中使用(Stallman[1994])；这在Davidson和Fraser(1984a,b)中有描述。其余的可以用C编译器编译的中间代码例子参Holub(1990)。

没有综合的最新优化技术参考，尽管标准参考Aho、Sethi和Ullman(1986)以及Fischer和LeBlanc(1991)包含了很好的总结。许多强大和实用的技术在ACM Programming languages Design and Implementation Conference Proceedings (先前称为Compiler Construction Conference)中发表，这是作为ACM SIGPLAN Notices的一部分出现的。额外的优化技术来源参见ACM Principles of Programming Languages Conference Proceedings 和ACM Transactions on Programming Languages and Systems。

附录A 编译器设计方案

本章要点

- C - 惯用的词法
- C - 语言的Tiny Machine运行时环境
- C - 的语法和语义
- 使用C - 和TM的编程设计
- C - 的程序例子

这里定义了一个编程语言称作C - Minus (或简称为C -), 这是一种适合编译器设计方案的语言, 它比TINY语言更复杂, 包括函数和数组。本质上它是C的一个子集, 但省去了一些重要的部分, 因此得名。这个附录由5小节组成。首先, 我们列出了语言惯用的词法, 包括语言标记的描述。其次, 给出了每个语言构造的BNF描述, 同时还有相关语义的英语描述。在A.3节, 给出了C - 的两个示例程序。再者, 描述了C - 的一个Tiny Machine运行时环境。最后一节描述了一些使用C - 和TM的编程设计方案, 适合于一个编译器教程。

A.1 C - 惯用的词法

1. 下面是语言的关键字：

```
else if int return void while
```

所有的关键字都是保留字, 并且必须是小写。

2. 下面是专用符号：

```
+ - * / < <= > >= == != = ; , ( ) [ ] { } /* */
```

3. 其他标记是ID和NUM, 通过下列正则表达式定义：

```
ID = letter letter*
NUM = digit digit*
letter = a|..|z|A|..|Z
digit = 0|..|9
```

小写和大写字母是有区别的。

4. 空格由空白、换行符和制表符组成。空格通常被忽略, 除了它必须分开 ID、NUM关键字。
5. 注释用通常的C语言符号 /*...*/ 围起来。注释可以放在任何空白出现的位置 (即注释不能放在标记内) 上, 且可以超过一行。注释不能嵌套。

A.2 C - 的语法和语义

C - 的BNF语法如下：

1. *program* *declaration-list*
2. *declaration-list* *declaration-list declaration* | *declaration*
3. *declaration* *var-declaration* | *fun-declaration*

4. var-declaration type-specifier **ID**; | type-specifier **ID** [**NUM**] ;
5. type-specifier **int** | **void**
6. fun-declaration type-specifier **ID** (params) | compound-stmt
7. params params-list | **void**
8. param-list param-list , param | param
9. param type-specifier **ID** | type-specifier **ID** []
10. compound-stmt { local-declarations statement-list }
11. local-declarations local-declarations var-declaration | empty
12. statement-list statement-list statement | empty
13. statement expression-stmt | compound-stmt | selection-stmt
 | iteration-stmt | return-stmt
14. expression-stmt expression ; | ;
15. selection-stmt **if** (expression) statement
 | **if** (expression) statement **else** statement
16. iteration -stmt **while** (expression) statement
17. return -stmt **return** ; | **return** expression ;
18. expression var = expression | simple-expression
19. var **ID** | **ID** [expression]
20. simple-expression additive-expression relop additive-expression
 | additive -expression
21. relop <= | < | > | >= | == | !=
22. additive-expression additive-expression addop term | term
23. addop + | -
24. term term mulop factor | factor
25. mulop * | /
26. factor (expression) | var | call | **NUM**
27. call **ID** (args)
28. args arg-list | empty
29. arg-list arg-list , expression | expression

对以上每条文法规则，给出了相关语义的简短解释。

1. *program* *declaration-list*
2. *declaration-list* *declaration-list declaration* | *declaration*
3. *declaration* *var-declaration* | *fun-declaration*

程序由声明的列表(或序列)组成, 声明可以是函数或变量声明, 顺序是任意的。至少必须有一个声明。接下来是语义限制(这些在C中不会出现)。所有的变量和函数在使用前必须声明(这避免了向后backpatching引用)。程序中最后的声明必须是一个函数声明, 名字为**main**。注意, C - 缺乏原型, 因此声明和定义之间没有区别(像C一样)。

4. *var-declaration* *type-specifier* **ID** ; | *type-specifier* **ID** [**NUM**] ;
5. *type-specifier* **int** | **void**

变量声明或者声明了简单的整数类型变量，或者是基类型为整数的数组变量，索引范围从 0 到 `NUM - 1`。注意，在 C - 中仅有的基本类型是整型和空类型。在一个变量声明中，只能使用类型

指示符 **int**。 **void** 用于函数声明(参见下面)。也要注意, 每个声明只能声明一个变量。

6. *fun-declaration* *type-specifier ID (params) compound-stmt*

7. *params* *param-list | void*

8. *param-list* *param-list , param | param*

9. *param* *type-specifier ID | type-specifier ID []*

函数声明由返回类型指示符、标识符以及在圆括号内的用逗号分开的参数列表组成, 后面跟着一个复合语句, 是函数的代码。如果函数的返回类型是 **void**, 那么函数不返回任何值(即是一个过程)。函数的参数可以是 **void** (即没有参数), 或者一系列描述函数的参数。参数后面跟着方括号是数组参数, 其大小是可变的。简单的整型参数由值传递。数组参数由引用来传递(也就是指针), 在调用时必须通过数组变量来匹配。注意, 类型“函数”没有参数。一个函数参数的作用域等于函数声明的复合语句, 函数的每次请求都有一个独立的参数集。函数可以是递归的(对于使用声明允许的范围)。

10. *compound-stmt* { *local-declarations statement-list* }

复合语句由用花括号围起来的一组声明和语句组成。复合语句通过用给定的顺序执行语句序列来执行。局部声明的作用域等于复合语句的语句列表, 并代替任何全局声明。

11. *local-declarations* *local-declarations var-declaration | empty*

12. *statement-list* *statement-list statement | empty*

注意声明和语句列表都可以是空的(非终结符 *empty* 表示空字符串, 有时写作 ϵ 。)

13. *statement* *expression-stmt*
 | *compound-stmt*
 | *selection-stmt*
 | *iteration-stmt*
 | *return-stmt*

14. *expression-stmt* *expression ; | ;*

表达式语句有一个可选的且后面跟着分号的表达式。这样的表达式通常求出它们一方的结果。因此, 这个语句用于赋值和函数调用。

15. *selection-stmt* **if** (*expression*) *statement*
 | **if** (*expression*) *statement* **else** *statement*

if语句有通常的语义: 表达式进行计算; 非 0 值引起第一条语句的执行; 0 值引起第二条语句的执行, 如果它存在的话。这个规则导致了典型的悬挂 **else** 二义性, 可以用一种标准的方法解决: **else** 部分通常作为当前 **if** 的一个子结构立即分析(“最近嵌套”非二义性规则)。

16. *iteration-stmt* **while** (*expression*) *statement*

while 语句是 C 中唯一的重复语句。它重复执行表达式, 并且如果表达式的求值为非 0, 则执行语句, 当表达式的值为 0 时结束。

17. *return-stmt* **return** ; | **return** *expression* ;

返回语句可以返回一个值也可无值返回。函数没有说明为 **void** 就必须返回一个值。函数声明为 **void** 就没有返回值。 **return** 引起控制返回调用者(如果它在 **main** 中, 则程序结束)。

18. *expression* *var = expression | simple-expression*

19. *var* *ID | ID [expression]*

表达式是一个变量引用, 后面跟着赋值符号(等号)和一个表达式, 或者就是一个简单的表达式。赋值有通常的存储语义: 找到由 *var* 表示的变量的地址, 然后由赋值符右边的子表达式

进行求值，子表达式的值存储到给定的地址。这个值也作为整个表达式的值返回。 *var* 是简单的(整型)变量或下标数组变量。负的下标将引起程序停止(与C不同)。然而，不进行下标越界检查。

*var*表示C - 比C的进一步限制。在C中赋值的目标必须是左值(**l-value**)，左值是可以由许多操作获得的地址。在C - 中唯一的左值是由 *var*语法给定的，因此这个种类按照句法进行检查，代替像C中那样的类型检查。故在C - 中指针运算是禁止的。

20. *simple-expression* *additive-expression relop additive-expression*
 | *additive -expression*

21. *relop* <= | < | > | >= | == | !=

简单表达式由无结合的关系操作符组成(即无括号的表达式仅有一个关系操作符)。简单表达式在它不包含关系操作符时，其值是加法表达式的值，或者如果关系算式求值为 *true*，其值为1，求值为*false*时值为0。

22. *additive-expression* *additive-expression addop term* | *term*

23. *addop* + | -

24. *term* *term mulop factor* | *factor*

25. *mulop* * | /

加法表达式和项表示了算术操作符的结合性和优先级。符号表示整数除；即任何余数都被截去。

26. *factor* (*expression*) | *var* | *call* | **NUM**

因子是围在括号内的表达式；或一个变量，求出其变量的值；或者一个函数调用，求出函数的返回值；或者一个**NUM**，其值由扫描器计算。数组变量必须是下标变量，除非表达式由单个**ID**组成，并且以数组为参数在函数调用中使用(如下所示)。

27. *call* **ID** (*args*)

28. *args* *arg-list* | *empty*

29. *arg-list* *arg-list* , *expression* | *expression*

函数调用的组成是一个**ID**(函数名)，后面是用括号围起来的参数。参数或者为空，或者由逗号分割的表达式列表组成，表示在一次调用期间分配的参数的值。函数在调用之前必须声明，声明中参数的数目必须等于调用中参数的数目。函数声明中的数组参数必须和一个表达式匹配，这个表达式由一个标识符组成表示一个数组变量。

最后，上面的规则没有给出输入和输出语句。在C - 的定义中必须包含这样的函数，因为与C不同，C - 没有独立的编译和链接工具；因此，考虑两个在全局环境中预定义的函数，好像它们已进行了声明：

```
int input(void) {...}
void output(int x) {...}
```

*input*函数没有参数，从标准输入设备(通常是键盘)返回一个整数值。*output*函数接受一个整型参数，其值和一个换行符一起打印到标准输出设备(通常是屏幕)。

A.3 C - 的程序例子

下面的程序输入两个整数，计算并打印出它们的最大公因子。

```
/* A program to perform Euclid's
   Algorithm to compute gcd. */
```

```

int gcd (int u, int v)
{ if (v == 0) return u ;
  else return gcd(v,u-u/v*v);
  /* u-u/v*v == u mod v */
}

void main(void)
{ int x; int y;
  x = input(); y = input();
  output(gcd(x,y));
}

```

下面的程序输入10个整数的列表，对它们进行选择排序，然后再输出：

```

/* A program to perform selection sort on a 10
   element array. */

int x[10];

int minloc ( int a[], int low, int high )
{ int i; int x; int k;
  k = low;
  x = a[low];
  i = low + 1;
  while (i < high)
  { if (a[i] < x)
    { x = a[i];
      k = i; }
    i = i + 1;
  }
  return k;
}

void sort ( int a[], int low, int high )
{ int i; int k;
  i = low;
  while (i < high-1)
  { int t;
    k = minloc (a,i,high);
    t = a[k];
    a[k] = a[i];
    a[i] = t;
    i = i + 1;
  }
}

void main (void)
{ int i;
  i = 0;
  while (i < 10)
  { x[i] = input;
    i = i + 1;
  }
}

```

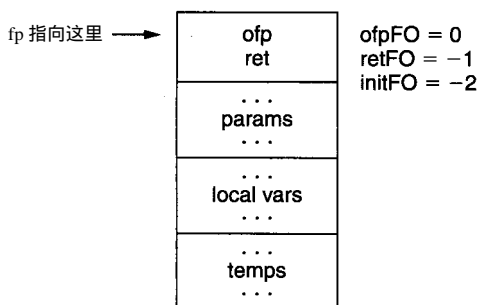
```

sort (x,0,10);
i = 0;
while (i < 10)
{ output(x[i]);
  i = i + 1;
}

```

A.4 C - 语言的Tiny Machine运行时环境

下面的描述采用了8.7节给出的Tiny Machine知识和第7章基于栈的运行时环境的知识。因为C - (与TINY不同)有递归过程,运行时环境必须是基于栈的。环境的组成部分有在 dMem顶部的全局区和在它下面的栈,朝下向 0增长。因为C - 不包含指针或动态分配,因此就不需要堆(heap)。在C - 中每个活动记录(或栈结构)的组成如下



这里,fp是当前结构指针(current frame pointer),为便于访问保存在一个寄存器中。ofp(旧结构指针)是正文第7章中讨论的控制链(control link)。在FO(结构偏移)右端的常数是每个存储的指示值的偏移量。值 initFO是在一个活动记录中存储区开始的参数和变量的偏移量。因为Tiny Machine不包含栈指针,对活动记录中所有字段的引用都使用带负结构偏移的 fp。

例如,如果有下列C - 函数声明:

```

int f(int x, int y)
{ int z;
  ...
}

```

那么x、y和z必须在当前结构中分配,f程序体代码产生的结构起始偏移量是-5(x、y和z各占一个地址,活动记录的簿记信息占两个地址)。x、y和z的偏移分别是-2、-3和-4。

在存储器中全局引用可以用绝对地址找到。然而,像TINY一样,我们更愿意从一个寄存器的偏移量引用这些变量。通过保存一个固定的寄存器实现这一点,称作gp,它总是指向最大的地址。因为TM模拟器在执行开始之前把这个地址存储到地址0,启动时gp可以从地址0装入,下面是初始化运行时环境的标准开始序列:

```

0: LD gp, 0(ac) * load gp with maxaddress
1: LDA fp, 0(gp) * copy gp to fp
2: ST ac, 0(ac) * clear location 0

```

函数调用也要求在一个调用序列中使用函数体的开始代码地址。我们也希望使用pc的当前值执行相对转移来调用函数而不是直接转移(这将使代码潜在地可重定位)。程序code.h/

code.c中的实用过程emitRAbs可以用于这个目的(它接受绝对代码地址,并通过使用当前的代码产生地址使其相对化)。

例如,假设要调用一个函数,其代码起始地址是27,当前的地址是42。那么代替产生绝对转移

```
42: LDC pc, 27(*)
```

我们将产生

```
42: LDA pc, -16(pc)
```

这是因为 $27 - (42 + 1) = -16$ 。

1) 调用序列 调用者和被调用者之间的合理划分是:使调用者除了在 retFO地址存储返回指针外,还在新的结构中存储参数的值并创建新的结构。代替存储返回指针本身,调用者把它留在ac寄存器中,被调用者把它存储进新的结构。因此,每个函数体必须从在(现在当前的)结构中存储值的代码开始:

```
ST ac, retFO(fp)
```

这在每个调用点保存一条指令。在返回时,每个函数通过执行指令

```
LD pc, retFO(fp)
```

用这个返回地址装入pc。相应地,调用者逐个计算参数,在新结构压栈之前把它们压进栈中相应的位置。调用者也必须先把当前的fp保存进结构的ofpFO处。从被调用者返回后,通过把旧的fp装入fp,调用者丢弃新结构。因此,对有两个参数的函数的调用将产生下列代码:

```
<code to compute first arg>
ST ac, frameoffset+initFO (fp)
<code to compute second arg>
ST ac, frameoffset+initFO-1 (fp)
ST fp, frameoffset+ofpFO (fp) * store current fp
LDA fp, frameoffset(fp) * push new frame
LDA ac,l(pc) * save return in ac
LDA pc, ...(pc) * relative jump to fuction entry
LD fp, ofpFO(fp) * pop current frame
```

2) 地址计算 因为变量和下标数组都允许出现在赋值表达式的左边,所以在编译期间必须区分地址和值。例如,在语句

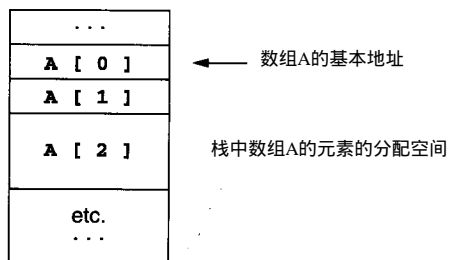
```
a[i] := a[i+1];
```

中,表达式a[i]指的是a[i]的地址,而表达式a[i+1]指的是a在地址i+1处的值。这个区分可以对cGen过程使用一个isAddress参数来实现。当这个参数为真时,cGen产生的代码计算变量的地址,而不是值。对于简单变量的情况,这意味着加上 gp(全局变量)或fp(局部变量)的偏移量并把结果装入到ac:

```
LDA ac, offset(fp) ** put address of local var in ac
```

对于数组变量的情况,这意味着加上相对于数组基地址的索引值,并把结果装入到ac,如下所述。

3) 数组 在栈中数组的分配从当前结构偏移量开始,按下标增长的顺序在存储器中向下延伸,如下所示:



注意，数组的地址通过从基地址中减去索引值计算。

当一个数组传递给函数时，仅传递基地址。基元素区域的分配只进行一次，并在数组生存期间保持固定。函数参数不包括数组的实际元素，仅仅是地址。因此，数组参数是引用参数。当数组参数在函数内部引用时这将引起异常，因为在存储器中保存的必须看成是它们的基地址而不是值。因此，数组参数计算基地址时使用LD操作代替LDA。

A.5 使用C - 和TM的编程设计

基于本书中讨论的TINY编译器(其清单在附录B中)，对于一个学期编译课程来说，要求把一个C - 语言的完整的编译器作为设计不是没有道理。这可以进行一些调整，当研究了相关的理论后实现编译器的每个阶段。另一方面，C - 编译器的一个或多个部分可以由导师提供，要求学生完成剩余的部分。当时间较短(如1/4学年)或者学生要产生“实际”机器的汇编代码，如Sparc或PC(在代码生成阶段要求更多的细节)，这就特别有用。对于仅实现C - 编译器的一部分这就不怎么有用，因为各部分之间的相互作用和代码测试的能力被限制了。下列分列的任务清单提供了一种安排，要注意每个任务与其他任务都不是独立的，最好完成所有的任务以获得完整的编写编译器的经验。

设计

1. 实现适合于C - 的一个符号表。要求表结构结合作用域信息，用于当各个独立的表链接到一起，或者有一个删除机制，用基于栈的方式操作，如第6章所述。
2. 实现一个C - 扫描器，或者像DFA用手工进行，或者使用Lex，如第2章所述。
3. 设计一个C - 语法树结构，适合于用分析器产生。
4. 实现一个C - 分析器(这要求一个C - 扫描器)，或者使用递归下降用手工进行，或者使用Yacc，如第4、5章所述。分析器要产生合适的语法树(见设计3)。
5. 实现C - 的语义分析器。分析器的主要要求是，除了在符号表中收集信息外，在使用变量和函数时完成类型检查。因为没有指针或结构，并且仅有的基本类型是整型，类型检查器需要处理的类型是空类型、整型、数组和函数。
6. 实现C - 的代码产生器，其根据是前一节描述的运行时环境。